The CSD Python API Documentation

version 1.0.0

November 06, 2015

CONTENTS

THE CSD PYTHON API

Conditions of Use

The Cambridge Structural Database System (CSD-System) comprising all or some of the following:

The Cambridge Structural Database System (CSD-System) comprising all or some of the following: ConQuest, Quest, PreQuest, deCIFer, Mercury, (Mercury CSD and Solid Form module [formerly known as the Materials module of Mercury], Mercury DASH), VISTA, Mogul, IsoStar, DASH, SuperStar, web accessible CSD tools and services, WebCSD, CSD Java sketcher, CSD data file, CSD-UNITY, CSD-MDL, CSD-SDfile, CSD data updates, sub files derived from the foregoing data files, documentation and command procedures, test versions of any existing or new program, code, tool, data files, sub-files, documentation or command procedures which may be available from time to time (each individually a Component) is a database and copyright work belonging to the Cambridge Crystallographic Data Centre (CCDC) and its licensors and all rights are protected. Use of the CSD-System is permitted solely in accordance with a valid Licence of Access Agreement or Products Licence and Support Agreement and all Components included are proprietary. When a Component is supplied independently of the CSD-System its use is subject to the conditions of the separate licence. All persons accessing the CSD-System or its Components should make themselves aware of the conditions contained in the Licence of Access Agreement or Products Licence and Support Agreement or the relevant licence.

In particular:

- The CSD-System and its Components are licensed subject to a time limit for use by a specified organisation at a specified location.
- The CSD-System and its Components are to be treated as confidential and may NOT be disclosed or re-distributed in any form, in whole or in part, to any third party.
- Software or data derived from or developed using the CSD-System may not be distributed without prior written approval of the CCDC. Such prior approval is also needed for joint projects between academic and for-profit organisations involving use of the CSD-System.
- The CSD-System and its Components may be used for scientific research, including the design of novel compounds. Results may be published in the scientific literature, but each such publication must include an appropriate citation as indicated in the Schedule to the Licence of Access Agreement or Products Licence and Support Agreement and on the CCDC website.
- No representations, warranties, or liabilities are expressed or implied in the supply of the CSD-System or its Components by CCDC, its servants or agents, except where such exclusion or limitation is prohibited, void or unenforceable under governing law.

Licences may be obtained from:

Cambridge Crystallographic Data Centre
12 Union Road
Cambridge CB2 1EZ, United Kingdom

Web: http://www.ccdc.cam.ac.uk
Telephone: +44-1223-336408
Email: admin@ccdc.cam.ac.uk

Release notes

Overview

The Cambridge Structural Database (CSD) is a highly curated and comprehensive repository of organic and organo-metallic crystal structures and is an essential resource to scientists around the world.

The Cambridge Structural Database System (CSDS) is a powerful and highly flexible suite of software tools and structural knowledge-bases. The CSDS enables exploration and application of the knowledge contained within more than 800,000 crystal structures.

The CSD Python API has been developed to make the CSD data and CSDS functionality accessible in a programmatic fashion. The aim is to facilitate integration with in-house work-flows and 3rd party applications. In addition, the CSD Python API can be used to perform activities not currently possible through the graphical interfaces. It is a platform for innovation.

Searchable documentation is available on at http://www.ccdc.cam.ac.uk/docs/csd_python_api/

An open community forum focused on using the CSD Python API is available at http://www.ccdc.cam.ac.uk/forum/csd_python_api/ and you are encouraged to visit this forum in the first instance if you require help or inspiration. We are also grateful for any feedback from your experiences with the CSD Python API. Alternatively, any feedback on the CSD Python API may be sent to support@ccdc.cam.ac.uk.

Citing the CSD Python API

When publishing works that benefited from the CSD Python API, please consider using the following citation:

> "The Cambridge Structural Database: a quarter of a million crystal structures and rising"
> F. H. Allen, Acta Cryst., B58, 380-388, 2002
> DOI: 10.1107/S0108768102003890.

For further citation advice, refer to your download agreement and visit http://www.ccdc.cam.ac.uk/support/product_references/

Licensed Features

In version 1.0 of the CSD Python API, some features are under development and are, currently, available only to associated collaborators. Some features are conditionally available depending on the user's CSD licence.

API Feature	All Users	CSD-Materials	CSD-Discovery	Associated Collaborators
IO	y	y	y	y
Entry, Crystal*1, Molecule	y	y	y	y
Search	y	y	y	y
Interaction	y	y	y	y
Descriptors*2	y	y	y	y
Diagram	y	y	y	y
Conformer*3	y	y	y	y
Screening				y
Docking				y

*1 The crystal packing similarity API is available to CSD-Materials users.

*2 The powder pattern simulation and comparison API is available to CSD-Materials users.

*3 The Conformer Generation API is available to CSD-Materials and CSD-Discovery users.

CSD-Enterprise users have access to CSD-Discovery and CSD-Materials feature sets.

Change Log

1.0.0

Backwards incompatible changes

None

Deprecations

None

Major new features

- updated for CSDS 2016
- there is now an API for docking ligands into proteins, using GOLD. This is currently available only to collaborators.
- **`ccdc.molecule.Molecule`** can now determine intramolecular hydrogen bonds and close contacts.
- **`ccdc.molecule.Atom.is_chiral`** and **`ccdc.molecule.Atom.chirality`** have been extensively revised to give more accurate determination of R/S chirality including the determination of para-chiral centres (whose chirality is determined solely by the chirality of other atoms). Note that structures with pi-bonds will not support the determination of chirality.
- **`ccdc.descriptors.MolecularDescriptors`** has new methods to define geometric objects from atom and ring coordinates.
- **`ccdc.descriptors.GeometricDescriptors`** is new and provides methods to define vectors and planes from points, and to calculate geometric relationships between them. See *Molecular geometry* for details.

Minor new features

Entry

- **`ccdc.entry.Entry.cross_references`** gives a tuple of **`ccdc.entry.Entry.CrossReference`** instances. These provide cross-refeerences between entries of the CSD.
- a **`ccdc.io.EntryReader`** of a mol2 format entry will now extract SDFile-like tags from the Mol2Comments and place them in an attributes dictionary. The EntryWriter will write these attributes.
- a **`ccdc.io.EntryReader`** of a mol2 format entry will now extract Mol2 format atom sets and place them in a dictionary attribute **`ccdc.entry.Entry.atom_sets`** where found. The EntryWriter will write atom sets if the above attribute is set.

Crystal

- **`ccdc.crystal.Crystal.molecule`** is now writable. A molecule may be extracted, edited and replaced in the crystal.

Molecule

- **`ccdc.molecule.Atom.sybyl_type`** gives the Sybyl atom type.
- **`ccdc.molecule.Bond.sybyl_type`** gives the Sybyl bond type.
- **`ccdc.molecule.Atom.vdw_radius`** gives the Van der Waals radius of the atom.
- **`ccdc.molecule.Molecule.from_string()`** now autodetects the file format of the string argument. The same is true of **`ccdc.crystal.Crystal.from_string()`** and **`ccdc.entry.Entry.from_string()`**.

Search

- search filters provided by **`ccdc.search.Search.Settings`** are now applied to **`ccdc.search.TextNumericSearch`**, **`ccdc.search.SimilaritySearch`** and **`ccdc.search.ReducedCellSearch`** searches.
- the **`ccdc.search.Search.Settings.no_errors`** attribute is now compatible with that used in ConQuest.
- the **`ccdc.search.Search.Settings.no_ions`** attribute has been provided.
- there is a new **`ccdc.search.SubstructureSearch.Screen`** which can be used to speed up substructure searches in multi-molecule mol2 or sdf format files. See *Substructure Screens* for details.
- similarly there is a new **`ccdc.search.SimilaritySearch.Screen`** for use when performing multiple similarity searches on a reader.

IO

- gcd files may now use an arbitrary database as the source of entries. Use the form io.EntryReader(gcd_file, source_database). This will also work with lists of identifiers.

0.7.0

Backwards incompatible changes

None

Deprecations

- The property **`ccdc.entry.jds_deposition_number`** is deprecated. This is a historical journal deposition number and has since been superseded by CCDC numbers. The method **`ccdc.search.TextNumericSearch.add_jds_deposition_number()`** is similarly deprecated.

Major new features

- There is a new class, **`ccdc.descriptors.MolecularDescriptors.MaximumCommonSubstructure`** which may be used to calculate the maximum common substructure of two **`ccdc.molecule.Molecule`** instances.

Minor new features

Entry

- **`ccdc.entry.Entry.remarks`** gives access to editorial comments on a database entry.
- **`ccdc.entry.Entry.pressure`** gives access to the experimental pressure of a crystallisation, where known.
- **`ccdc.entry.Entry.is_powder_study`** indicates whether the experimental determination was performed in a powder study

Crystal

- The reduced cell of a crystal is now available as **`ccdc.crystal.Crystal.reduced_cell`**. The value is an instance of **`ccdc.crystal.Crystal.ReducedCell`**.

IO

- res format (SHELX) files may now be read and written through the standard io classes.

Search

- There is a method for writing ConQuest to Mercury interchange files using **`ccdc.search.SubstructureSearch.SubstructureHitList.write_c2m_file()`**. This allows substructure search results to be visualised in the data analysis module of Mercury.
- A substructure search hit has a new method, **`ccdc.search.SubstructureSearch.SubstructureHit.match_symmetry_operators()`** which will return the symmetry operators applied to atoms to perform the match.
- There is a new method, **`ccdc.search.Search.Settings.test()`** which will determine whether an **`ccdc.entry.Entry`**, **`ccdc.crystal.Crystal`** or **`ccdc.molecule.Molecule`** satisfies the requirements of the Settings class.
- **`ccdc.search.Search.Settings.no_disorder`** now may take any of three values: None (or anything which evaluates to False) to indicate no filtering of any disordered structures; 'all' to indicate filtering of structures with any disordered atoms; 'Non-hydrogen' or any string apart from 'all' to indicate filtering of structures with heavy atom disorder. This last is compatible with the ConQuest 'no disorder' selector.

GeometryAnalyser

- **`ccdc.conformer.GeometryAnalyser.Analysis.generalised`** now reports whether generalisation was needed to find enough observations.

Screener

- The CSD field based ligand screener, within the **`ccdc.screening`** module, in this release has undergone further development and results will differ those generated from version 0.6.
- The score threshold and the cluster radius used to generate the fitting points from the grids may be defined through **`ccdc.screening.Screener.Settings.fitting_points_threshold`** and **`ccdc.screening.Screener.Settings.fitting_points_cluster_radius`**, respectively.
- During the search the conformer selection can be biased towards the top ones in the supplied list of conformers according to **`ccdc.screening.Screener.Settings.bias_conformer_selection`**

Examples

- maximum_common_substructure.py shows a similarity search followed by maximum common substructure search.
- filter_csd.py shows how an iteration over the entries of the CSD can omit entries on a number of criteria.
- simple_report.py shows how python's format() method may be used to write HTML reports on a CSD entry.

Bug fixes

None

0.6.0

Backwards incompatible changes

- Previously molecules constructed from crystals or entries read from the CSD would contain disordered atoms, even where CCDC editorial staff were able to resolve the disorder. This is no longer the case. Instead the properties **`ccdc.entry.Entry.molecule`** and **`ccdc.crystal.Crystal.molecule`** will suppress disordered atoms. The properties **`ccdc.entry.Entry.disordered_molecule`** and **`ccdc.crystal.Crystal.disordered_molecule`** may be used to retrieve the disordered atoms.
- The default file format for **`ccdc.io.MoleculeWriter`**, **`ccdc.io.CrystalWriter`** and **`ccdc.io.EntryWriter`** has been changed from 'aser' to 'mol2', in cases where no file suffix or explicit format has been specified. This is more intuitive when writing to sys.stdout.

- The deprecated modules **`ccdc.mogul`** and **`ccdc.isostar`** are removed.
- **`ccdc.conformer.GeometryAnalyser.Analysis`** no longer support z-score where the analysis type is 'torsion' or 'ring'. It is a meaningless statistic for these distributions. Instead the value None is returned.

Deprecations

- The properties **`ccdc.conformer.GeometryAnalyser.Settings.GeometrySettings.usual_density_threshold`** and **`ccdc.conformer.GeometryAnalyser.Settings.GeometrySettings.usual_zscore_threshold`** have been deprecated in favour of the more descriptive **`ccdc.conformer.GeometryAnalyser.Settings.GeometrySettings.local_density_threshold`** and **`ccdc.conformer.GeometryAnalyser.Settings.GeometrySettings.zscore_threshold`**. The former pair will be removed in the next release.

Major new features

- Substructure searches now support a set of filters to control acceptable results, for example, only organic structures, structures with an R-factor less than an arbitrary limit, or a set of unacceptable elements. See *Substructure searching* for examples and **`ccdc.search.Search.Settings`** for details.
- Crystal packing similarities may now be calculated using **`ccdc.crystal.PackingSimilarity`**. See *Crystal packing similarity* for examples. *Feature under development – currently available only to associated collaborators.*
- **`ccdc.molecule.Molecule`** now has an editing API, allowing the construction and mutation of molecules. See *Editing molecules* for examples.
- **`ccdc.molecule.Molecule`** now supports ring and conjugated bond analysis. See *Rings*
- Added the **`ccdc.screening`** module for CSD-driven field-based ligand screening. See *Field-based virtual screening*. *Feature under development – currently available only to associated collaborators.*

Minor new features

Geometry Analyser

- **`ccdc.conformer.GeometryAnalyser`** will now analyse the geometries of molecules, bonds, angles and torsions of atoms with no 3D information. It will not be able to analyse rings with no 3D information.
- **`ccdc.conformer.GeometryAnalyser.fragment_identifier()`** may be used to extract a unique identifier for a particular type of fragment.
- **`ccdc.conformer.GeometryAnalyser.Settings.GeometrySettings`** now has properties to inspect and control the precise nature of the classification of an observation as unusual.

Diagrams

- **`ccdc.diagram.DiagramGenerator`** will now generate images as SVG or PIL images.

Search

- **`ccdc.search.TextNumericSearch`** has a new property, queries, which will give a human-readable representation of the queries added to the search instance.

Entry

- **`ccdc.entry.Entry`** has the property radiation_source to express the experimental radiation probe used in the determination of the crystal.

- **ccdc.entry.Entry** has the property is_polymeric to allow filtering of the CSD on polymeric structures.
- **ccdc.entry.Entry** and **ccdc.crystal.Crystal** may now be compared for equality (based on its identifier) and hashed, allowing use as a key in a dictionary or set.

Molecule

- **ccdc.molecule.Molecule** can now assign formal charges to atoms based on a correct bond-typing and protonation of the molecule.
- There is a new method, **ccdc.molecule.Molecule.kekulize()** which will convert aromatic bonds to alternating single and double bonds.
- **ccdc.molecule.Molecule** now has the property **ccdc.molecule.Molecule.is_polymeric** to reflect whether the molecule contains polymeric bonds.
- **ccdc.molecule.Molecule** now has methods **ccdc.molecule.Molecule.atom()** and **ccdc.molecule.Molecule.bond()** to allow access to atoms and bonds by atom labels.
- **ccdc.molecule.Molecule** now has convenience properties **ccdc.molecule.Molecule.heavy_atoms** and **ccdc.molecule.Molecule.is_3d** to provide a list of the heavy atoms of the molecule, and to determine whether the molecule's heavy atoms have 3d coordinates.
- **ccdc.molecule.Atom** now has a property, 'fractional_coordinates' to provide coordinates relative to the molecule's underlying crystal's unit cell.
- **ccdc.io** now supports reading CSD SQLite databases by file names with suffix '.sqlite'
- **ccdc.io** now supports reading and writing '.mol' files as an alternative suffix to '.sdf'
- **ccdc.io** reader and writer classes now report the location of their files through __str__ and __repr__ methods.
- **ccdc.molecule.Atom** now has attributes `bonds` denoting the bonds to this atom, and `neighbours` denoting the atoms to which the atom is bonded.
- **ccdc.molecule.Bond** now supports equality and hashing so may be put into a set or used as a dictionary key.
- **ccdc.molecule.Atom** now has attributes `chirality` to return the R/S chirality flag for the atom and `is_chiral` returning a bool.

Crystal

- **ccdc.crystal.Crystal** now has a `spacegroup_number_and_setting` attribute to provide details of the crystal's spacegroup.
- **ccdc.crystal.Crystal** has a new method to calculate a packing shell of a given number of molecules.
- **ccdc.crystal.Crystal** now reports its symmetry operations as a tuple of strings.
- **ccdc.crystal.Crystal** will report rotational and translational components of its symmetry operators.
- **ccdc.crystal.Crystal** now allows the generation of molecules generated by the crystal's symmetry operations.

0.5.0

Backwards incompatible changes

- The `add_refcode` function has been replaced by **ccdc.search.TextNumericSearch.add_identifier()**. Furthermore this function now only matches the current identifier of an entry. The previous behaviour, which also tried to find matches using any previous identifiers can be accessed using the **ccdc.search.TextNumericSearch.add_all_identifiers()** function.

Deprecations

- The **`ccdc.mogul`** module has been deprecated. Please use the **`ccdc.conformer.GeometryAnalyser`** class instead.
- The **`ccdc.isostar`** module has been deprecated. Please use the **`ccdc.interaction`** module instead.

Major new features

- Updated for CSDS 2015
- Ability to search through and work with multiple and in-house ASER databases *Working with multiple databases*
- Ability to highlight atom selections in 2D diagrams; see *Highlighting atom selections in diagrams*
- Ability to access the CSD curated diagram when generating images of CSD entries; see *Accessing diagrams from CSD entries*
- It is now possible to construct a molecule from scratch; see *Building molecules from scratch*
- It is now possible to generate powder patterns and to calculate powder pattern similarities; see *Powder patterns*
- It is now possible to perform reduced cell searches; see *Reduced cell searching*
- Entries without any 3D information are no longer ignored

Minor new features

- CIF (Crystallographic Information File) format is now supported by all the **`ccdc.io`** writers.
- **`ccdc.molecule.Molecule`** has several shortest path functions:
  - **`ccdc.molecule.Molecule.shortest_path()`**
  - **`ccdc.molecule.Molecule.shortest_path_atoms()`**
  - **`ccdc.molecule.Molecule.shortest_path_bonds()`**
- It is now possible to create searches using the XML format generated by the WebCSD Java sketcher
  - **`ccdc.search.SubstructureSearch.read_xml()`**
  - **`ccdc.search.SubstructureSearch.read_xml_file()`**
  - **`ccdc.search.SubstructureSearch.from_xml()`**
  - **`ccdc.search.SubstructureSearch.from_xml_file()`**
  - **`ccdc.search.SimilaritySearch.read_xml()`**
  - **`ccdc.search.SimilaritySearch.read_xml_file()`**
  - **`ccdc.search.SimilaritySearch.from_xml()`**
  - **`ccdc.search.SimilaritySearch.from_xml_file()`**
  - **`ccdc.search.TextNumericSearch.read_xml()`**
  - **`ccdc.search.TextNumericSearch.read_xml_file()`**
  - **`ccdc.search.TextNumericSearch.from_xml()`**
  - **`ccdc.search.TextNumericSearch.from_xml_file()`**
- Additions to **`ccdc.search.TextNumericSearch`**
  - **`ccdc.search.TextNumericSearch.add_ccdc_number()`**
  - **`ccdc.search.TextNumericSearch.add_source()`**
  - **`ccdc.search.TextNumericSearch.add_all_identifiers()`** checks both the current and any previous identifiers of an entry for a match

    - New `start_of_word` added as an option to the `mode` argument, which matches start of any word in a searched string
    - Old `start` option of the `mode` argument will now only match the start of the searched string
    - Added `ignore_non_alpha_num` argument
- Additions to the **`ccdc.entry.Entry`**
    - **`ccdc.entry.Entry.ccdc_number`**
    - **`ccdc.entry.Entry.source`**
    - **`ccdc.entry.Entry.previous_identifier`**
    - **`ccdc.entry.Entry.diagram_image()`** (curated 2D diagram image)

Bug fixes

- **`ccdc.entry.Entry.chemical_name`** and **`ccdc.entry.Entry.synonyms`** now return non-ascii characters as Unicode

0.4.0

Backwards incompatible changes

None

Major new features

- Access to the CCDC conformer generator and molecular minimiser. See the **`ccdc.conformer`** module. *Feature under development – currently available only to associated collaborators.*
- Access to the CCDC diagram generation functionality. See the **`ccdc.diagram`** module.
- Mogul analysis of individual fragments:
    - **`ccdc.mogul.Mogul.analyse_bond()`**
    - **`ccdc.mogul.Mogul.analyse_angle()`**
    - **`ccdc.mogul.Mogul.analyse_torsion()`**
    - **`ccdc.mogul.Mogul.analyse_ring()`**
- The entire CSD Python API is now unicode compatible

Minor new features

- Some tuple data now represented using the `namedtuple` collection
    - **`ccdc.entry.Entry.publication`**
    - **`ccdc.crystal.Crystal.cell_lengths`**
    - **`ccdc.crystal.Crystal.cell_angles`**
    - **`ccdc.molecule.Atom.coordinates`**
- Functions for reading from and writing to strings added for entries and crystals
    - **`ccdc.entry.Entry.to_string()`**
    - **`ccdc.entry.Entry.from_string()`**
    - **`ccdc.crystal.Crystal.to_string()`**
    - **`ccdc.crystal.Crystal.from_string()`**
- Function for reading in a Connser substructure search from a string

    - **ccdc.search.ConnserSubstructure.from_string()**
- It is now also possible to write out and read in **ccdc.entry.Entry** attributes as sdf tags to and from mol2 files. The attributes read in are dynamically added to a dictionary attribute named attributes.

Bug fixes

- Fixed a bug which meant that the **ccdc.mogul.MogulResult.histogram()** function was not returning correct data.

0.3.1

Backwards incompatible changes

- Renaming of Query classes in the **ccdc.search** module to Search classes
    - **ccdc.search.TextNumericSearch** supersedes TextNumericQuery
    - **ccdc.search.SimilaritySearch** supersedes SimilarityQuery
    - **ccdc.search.SubstructureSearch** supersedes SubstructureQuery
- Renaming of max_hits argument to max_hit_structures in the **ccdc.search.TextNumericSearch.search()**, **ccdc.search.SimilaritySearch.search()** and **ccdc.search.SubstructureSearch()** functions
- The argument signature of the **ccdc.crystal.Crystal.molecular_shell()** function has been updated. It now takes a tuple named distance_range that contains the minimum and maximum values as opposed to the two explicitly named arguments range_min and range_max.

Major new features

- **Support for 64-bit Python on Linux**
- **ccdc.search.SubstructureSearch** has expanded functionality
    - Ability to **measure** distances, angles and torsion angles in hit structures
    - Ability to **constrain** distances, angles and torsion angles in hit structures
    - It now provides the ability to add more than one substructure, which can be used to set up inter-molecular contact searches
- A number of IsoStar classes have been implemented. See **ccdc.isostar** for details.

Minor new features

- The **ccdc.io.EntryReader** class now supports the reading of CIF format files.
- The **ccdc.entry.Entry** class contains a dictionary-like member, 'attributes', which will contain CIF data members when the entry comes from a CIF database.
- **ccdc.crystal.Crystal** has new properties
    - **ccdc.crystal.Crystal.crystal_system**
    - **ccdc.crystal.Crystal.calculated_density**
- **ccdc.descriptors.MolecularDescriptors** has new static methods
    - **ccdc.descriptors.MolecularDescriptors.atom_distance()**
    - **ccdc.descriptors.MolecularDescriptors.atom_angle()**
    - **ccdc.descriptors.MolecularDescriptors.atom_torsion_angle()**
- **ccdc.molecule.Molecule**: has new functionality

  - A new property, **`ccdc.molecule.Molecule.formula`**
  - A new function **`ccdc.molecule.Molecule.to_string()`**
  - A new static method **`ccdc.molecule.Molecule.from_string()`**

  The latter two allow simple conversions between a molecule and a string representation.

- **`ccdc.search.SMARTSSubstructure`** may now take an extended SMARTS string containing an integer label on an atom prefixed by a colon. These labels may be used to specify measurements to be made on hit results.
- **`ccdc.utilities`** has a new class, **`ccdc.utilities.FileLogger`** which supports the context manager protocol, and will allow temporary redirection of messages.

Bug fixes

- There is a much larger number of open ASER database instances provided, and a memory leak of open ASER database instances has been fixed.
- **`ccdc.search.ConnserSubstructure`**: will raise an exception for a missing or empty file name parameter.

Known issues

If your problem isn't addressed below, we invite you to visit the Technical Help section of the CSD Python Community Forum at http://www.ccdc.cam.ac.uk/forum/csd_python_api/

UCS2 vs UCS4 compatible Python on Linux

As of version 0.4 the CSD Python API is unicode compatible. However, this means that one will require a Python installation with the same Universal Character Set (UCS).

The CSD Python API is compiled against a UCS4 enabled version of Python. If you use a Python that was compiled against UCS2 you may see the error below:

```
undefined symbol: _PyUnicodeUCS4_AsDefaultEncodedString
```

In this case you will need to compile a new version of Python with UCS4 enabled. On Linux this can be done by downloading the source for the latest Python 2.7 release and using the commands below:

```
$ ./configure --prefix=/home/user/MyPython --enable-unicode=ucs4
$ ./make
$ ./make install
```

Alternatively, please do get in contact with us and we can try to make a UCS2 compiled version of the CSD Python API available to you.

No version information available in libz.so.1

Depending on the version of Linux that you are using you may see an warning along the lines of:

```
/home/user/MyPython/python2.7/site-packages/ccdc/libz.so.1: no version information available (required by python))
```

To work around this please rename or remove the `/home/user/MyPython/python2.7/site-packages/ccdc/libz.so.1` file:

```
$ cd /home/user/MyPython/python2.7/site-packages/ccdc/
$ mv libz.so.1 backup.libz.so.1
```

IPython sqlite3.dll incompatibility

If you make use of the `ccdc` package in IPython notebooks, on Windows, you may see a `RuntimeError`:

```
RuntimeError: prepare_statement failed (code 1) - no such module: FTS3 - D:\my\csd\as534be_ASER.sqlite
```

This is due to IPython loading the `C:\Python27\DLLs\sqlite3.dll` before the **`ccdc`** module has the chance to load the `C:\Python27\Lib\site-packages\ccdc\sqlite3.dll`. The difference between these two DLLs is that the former lacks the FTS3 sqlite module extension, which is required by some parts of the **`ccdc`** module.

The simplest solution is to replace the former DLL with the version shipped with the CSD Python API.

ASER database handles

Creating more than 50 instances of an ASER database in the same Python session will cause an error. This is a known issue and can manifest itself when constructing large numbers of Mogul searchers. However, in practise it should not present a real problem as one is unlikely to want more than one instance of a Mogul object at once.

Diagram highlighting gives misleading results if used with shrunken symbols

When generating 2D diagrams of molecules and crystals it is recommended to set the **`ccdc.diagram.DiagramGenerator.Settings.shrink_symbols`** setting to `False` as it may otherwise give misleading images.

This issue does not manifest itself when using **`ccdc.entry.Entry`** where the 2D coordinates are taken from the CSD.

Diagram generation gives harmless error messages during tests on some systems

When generating 2D diagrams of molecules and crystals on some platforms, including Debian 7.8, various "Gtk-CRITICAL" messages are written to the stderr stream. This only occurs when the nose module is imported.

The generated images are correct and these messages may be safely ignored.

Incompatible Qt libraries with Anaconda Python

This problem might be seen when installing the pip package into an Anaconda Python environment. It is not seen with the conda package.

When installing the CSD Python API into Anaconda Python with PyQt and Qt already present the tests will fail to run. Errors such as the following `ImportError` will be reported:

```
ImportError: /home/user/python/anaconda/1.7.0/64/bin/../lib/libQt3Support.so.4: undefined symbol: _ZNK9QMenuItem10metaObjectEv
```

One solution is to uninstall the Qt related packages, or to reinstall a version compiled with the *--no-qt3support* option. A preferred solution is to use the conda installation package for the CSD Python API.

Problem with Max OS X XQuartz versions prior to 2.7.8

For Mac OS X users, if a script fails to run with an error containing a message similar to "Library not loaded: /usr/X11/lib/libfreetype.6.dylib", please ensure that your XQuartz installation is updated to at least v.2.7.8. There are known problems with earlier versions.

Dependent Numpy install fails with "error: Unable to find vcvarsall.bat"

On Windows, the install may stop due to failure in numpy installation. This is a results of the Visual C++ 2008 runtime libraries are not installed on the computer. These can be added through installation of the free Visual C++ 2008 Express Edition.

Installation notes

Support and Help

If you have any problems or suggestions please do get in touch with us at support@ccdc.cam.ac.uk.

An open community forum focused on using the CSD Python API is available at http://www.ccdc.cam.ac.uk/forum/csd_python_api/ and you are encouraged to visit this forum in the first instance if you require help or inspiration. We are also grateful for any feedback from your experiences with the CSD Python API.

Supported platforms

The majority of the development and testing was carried out on:

- Windows 7
- Linux RedHat Enterprise 6, 32-bit and 64-bit

However, some testing has been performed on other platforms and we believe that those listed below will present no difficulty in running.

- **Windows - Intel compatible, 32-bit and 64-bit**
    - Windows Vista, 7, 8 and 10
- **Linux - Intel compatible, 32-bit and 64-bit**
    - RedHat Enterprise 5, 6 and 7
    - SuSE Linux Enterprise [Desktop|Server] 11
    - Debian Stable 6, 7 and 8.
    - CentOS 5, 6 and 7
- OS X 10.9 and 10.10

Python version

- Windows - Python 2.7 **32-bit only**. The 64-bit Python is *not* supported on Windows.
- Linux - Python 2.7 32-bit and 64-bit
- OS X - Python 2.7 from python.org. The OS X pre-installed Python is *not* supported. We recommend the use of Anaconda Python for OS X.

Installation

Downloads

The downloads required for the CSD Python API are available from the CSDS download page:

- **CSD Python API - Python package**
    - pip package *or* conda package
- CSD Python API - CSD SQLite Database

Warning

If you previously installed a version of the CSD Python API earlier than 0.7, you must uninstall this before installing this release. Older files are not removed during the install process and these will prevent correct behaviour of the API.

Note

The CSD Python API may be installed using either *pip* or *conda*. Conda is a part of the Anaconda Python Distribution. The following instructions assume you are using pip. For Anaconda Python, following steps 1-3, below, and then:

```
``conda install csd-python-api-1.0.0-win32-py2.7-conda.tar.bz2``
```

Windows

1. If you have not yet installed the CSDS 2016 release, then you will need to do this first.
2. The CSDS 2016 release does not include the CSD SQLite database file, which you will need to install separately. To install the CSD SQLite database use the `CSDSQLite-2016-windows-installer.exe`

   This should automatically detect your CSDS installation and direct you to install in the CSD database directory. If it does not detect the directory then you can manually specify the correct directory, for example:

   ```
   C:\my\csds\install\CSD V5.37
   ```

   If you cannot install directly into the CSDS installation the CSD SQLite database can be installed elsewhere and an environment variable used to locate the CSDS data, for example:

   ```
   set CCDC_TOOLKIT_SQLITE_DATABASE=C:\my\csds\install\CSD V5.37\as537be_ASER.sqlite
   ```

   Note that the CSD SQLite database is a large compressed file and will probably take several minutes to unzip.
3. If you wish to make use of the CSD Python API molecular interaction interface you must set the CCDC_ISOSTAR_DATA_DIRECTORY environment variable to point to the installed data files. The CSDS 2016 installer includes the IsoStar files so the variable should be set to find the correct "isostar_files" folder as follows:

   ```
   set CCDC_ISOSTAR_DATA_DIRECTORY=C:\my\csds\install\CSD_System_2016\isostar_files
   ```
4. If you do not already have a Python 2.7 installation then please download and run the latest Windows installer from www.python.org/downloads. By default this will create:

   ```
      C:\Python27

   Make sure that your path variable includes your Python directory and
   your Python Scripts directory. For example by updating the PATH::

      set PATH=%PATH%;C:\Python27;C:\Python27\Scripts
   ```
5. If it is not already installed, install *pip*. pip is included as standard with recent Python versions.
6. Install the CSD Python API by running `pip install csd-python-api-1.0.0-win32-py2.7.zip`

Warning

If the installation fails to load the required modules with an error similar to 'Unable to load _UtilitiesLib' then it is likely that your computer does not contain the required Microsoft runtime libraries. These may be installed from: the Visual C++ Redistributable Package for Visual Studio 2013

Warning

pip automatically downloads additional packages such as NumPy. Some company firewalls block this, so you may need to configure an HTTPS proxy. Ask your systems administrator for more information.

Note

Several example scripts for the CSD Python API require the Python matplotlib package, as do some Mercury example scripts. Matplotlib may be installed with:

```
``pip install matplotlib``
```

Linux & OS X

1. If you have not yet installed the CSDS 2016 release, then you will need to do this first.
2. The CSDS 2016 release does not include the CSD SQLite database file, which you will need to install separately. To install the CSD SQLite database use the `CSDSQLite-2016-linux-installer.run` (Linux) or `CSDSQLite-2016-osx-installer.tar` (OS X).

   This can be installed into the CSD database directory, which you will need to specify manually, for example:

   ```
   /home/my/software/csds_v537/csd
   ```

   If you cannot install directly into the CSDS installation the CSD SQLite database can be installed elsewhere and an environment variable used to locate the CSD data, for example:

   ```
   $ export CCDC_TOOLKIT_SQLITE_DATABASE=/home/my/elsewhere/as537be_ASER.sqlite
   ```

   Note that the CSD SQLite database is a large compressed file and will probably take several minutes to unzip.
3. If you wish to make use of the API's molecular interaction interface you will need to set the CCDC_ISOSTAR_DATA_DIRECTORY environment variable to point to the installed data files. The CSDS 2016 installer includes the IsoStar files so the variable should be set as follows:

   ```
   $ export CCDC_ISOSTAR_DATA_DIRECTORY=/home/my/software/csds_v537/isostar_files
   ```
4. If you do not already have a Python 2.7 installation then please download and run the latest Linux installer tarball from www.python.org/downloads. To build and install locally:

   ```
   $ tar zxvf Python-2.7.9.tgz
   $ cd Python-2.7.9
   $ ./configure --prefix=/home/my/python2.7 --enable-unicode=ucs4
   $ make
   $ make install
   ```

5. **Linux Only** If you already have Python 2.7 installed please check that it is UCS-4 enabled. You may do this with:

```
$ python -c "import sys; print sys.maxunicode > 65536 and 'UCS-4' or 'UCS-2'"
```

> **Seealso**
>
> For more detail on how to find out if Python is compiled with UCS-2 or UCS-4 please see stackoverflow.com/questions/1446347

   If UCS-4 is not enabled you will need to install a UCS-4 enabled version of Python 2.7 (see step 4 above). Alternatively, please do get in contact with us and we can try to make a UCS-2 compiled version of the CSD Python API available to you.

6. If it is not already installed, install *pip*. pip is included as standard with recent Python versions.
7. Make sure you edit your .bashrc environment variables to use the new Python installation. Then to install the ccdc modules into your own Python 2.7 run on 32-bit Linux:

```
$ pip install csd-python-api-1.0.0-linux-py2.7.tar.gz
```

   Or, on 64-bit Linux:

```
$ pip install csd-python-api-1.0.0-linux-64-py2.7.tar.gz
```

   Or, on OS X:

```
$ pip install csd-python-api-1.0.0-mac-64-py2.7.tar.gz
```

> ***Warning***
>
> pip automatically downloads additional packages such as NumPy. Some company firewalls block this, so you may need to configure an HTTPS proxy. Ask your systems administrator for more information.

8. Finally, there are a few more environment variables that need to be set to enable the CSD Python API to talk to the installed CSD system. `CSDHOME` needs to point to the installation directory of the CSD-System as a whole, not the data:

```
$ export CSDHOME=/home/my/software/csds_v537/
```

   On Linux only, assumming that $PYTHONHOME stores the location of the correct Python installation:

```
$ export LD_LIBRARY_PATH=$PYTHONHOME/lib/python2.7/site-packages/ccdc/_lib:$LD_LIBRARY_PATH
```

   On Mac only:

```
$ export DYLD_LIBRARY_PATH=$PYTHONHOME/lib/python2.7/site-packages/ccdc/_lib
$ export DYLD_FRAMEWORK_PATH=$PYTHONHOME/lib/python2.7/site-packages/ccdc/_lib
```

   It is important that older versions of the CSD Python API are not additionally referenced within the `LD_LIBRARY_PATH` value (`DYLD_LIBRARY_PATH` on OS X).

   If you have installed any of the CSD components in non-standard locations it is possible to specify these locations using the environment variables listed below:

```
$ # Set the variables required to locate the CSD-System and bindings
$ export CCDC_TOOLKIT_ASER_DATABASE=/home/my/software/csds_v537/csd/as536be
$ #export CCDC_TOOLKIT_SQLITE_DATABASE=/home/my/elsewhere/as537be_ASER.sqlite
$ #export CCDC_ISOSTAR_DATA_DIRECTORY=/home/my/elsewhere/isostar_files
$ export CCDC_MOGUL_DATA=/home/my/software/csds_v537/data
$ export CCDC_MOGUL_ASER_DATABASE=${CCDC_TOOLKIT_ASER_DATABASE}
$ export CCDC_MOGUL_INITIALISATION_FILE=$PYTHONHOME/lib/python2.7/site-packages/ccdc/parameter_files/mogul.ini
```

Warning

On Mac OS X only, if a script fails to run with an error containing a message

similar to "Library not loaded: /usr/X11/lib/libfreetype.6.dylib", please ensure that your XQuartz installation is updated to at least v.2.7.8.

Note

Several example scripts for the CSD Python API require the Python matplotlib package, as do some Mercury example scripts. Matplotlib may be installed with:

```
``pip install matplotlib``
```

Testing Your Installation

Windows

1. Install *nose*:

   ```
   pip install nose
   ```

2. Extract the `tests` directory from the package zip file.
3. To run the CSD Python API unit tests execute the `run_tests.bat` script:

   ```
   run_tests.bat
   ```

   The expected output from this script is that all the tests pass with an OK, except for the specific cases noted:

4. When running the `testDescriptorsAPI` unit test the message:

   ```
   "ERROR: Molecule does not have sites for any atoms"
   ```

   is written to the test output. This is expected and is not a test failure.

5. On Windows XP the `testIgnoreLineNumbers` unit test will fail.
6. If Isostar or Mogul data sources were not defined in step 8, above, then some related test failures are expected.

Linux & OS X

1. Install *nose*:

   ```
   $ pip install nose
   ```

2. Extract the `tests` directory from the package tar.gz file.

3. Make the `run_tests.sh` script executable:

```
$ chmod +x run_tests.sh
```

4. To run the CSD Python API unit tests execute the `run_test.sh` script:

```
$ ./run_tests.sh
```

   The expected output from this script is that all the tests pass with an OK, except for the specific cases noted:

5. When running the `testDescriptorsAPI` unit test the message:

```
"ERROR: Molecule does not have sites for any atoms"
```

   is written to the test output. This is expected and is not a test failure.

Descriptive documentation

Quick primer to using the CSD Python API

Introduction

This section is a quick primer for getting familiar with the CSD Python API. It is not, however, a complete reference to the CSD Python API. If this is what you are interested in have a look at the *API documentation*.

Once you have read through this primer and want some more information on how to make use of the CSD Python API it is recommended that you have a look at the examples included in the *cookbook documentation* and make use of the *descriptive documentation*.

> ***Note***
>
> Example files referenced within this documentation can be found in the doc/example_files directory within the distributed archive.

Starting a Python interactive shell

This document will make use of the Python interactive shell:

```
bash-3.2$ python
Python 2.7.3 (default, Jun 21 2012, 13:00:25)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-52)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

> **Seealso**
>
> For more information on Python's interactive mode please see the Python on-line documentation docs.python.org/2/tutorial/interpreter.html#interactive-mode.

Accessing database entries, crystals and molecules from the CSD

The CSD Python API makes a distinction between database entries (**`ccdc.entry.Entry`**), crystals (**`ccdc.crystal.Crystal`**) and molecules (**`ccdc.molecule.Molecule`**). To illustrate the difference let us get access to the database entry, crystal and molecule of ABEBUF.

```
>>> from ccdc import io
>>> csd_reader = io.EntryReader('CSD')
>>> entry_abebuf = csd_reader.entry('ABEBUF')
>>> cryst_abebuf = csd_reader.crystal('ABEBUF')
>>> mol_abebuf = csd_reader.molecule('ABEBUF')
```

Molecules

A **`ccdc.molecule.Molecule`** contains molecular properties such as the molecular weight (**`ccdc.molecule.Molecule.molecular_weight`**) and descriptors such as whether or not the molecule is organic (**`ccdc.molecule.Molecule.is_organic`**).

```
>>> round(mol_abebuf.molecular_weight, 3)  # only want 3 significant figures
317.341
```

```
>>> mol_abebuf.is_organic
True
```

Seealso

In the example above we make use of Python's built in `round` function. For more information on the `round` function please see the Python on-line documentation docs.python.org/2/library/functions.html#round.

The molecule also contains a **`ccdc.molecule.Molecule.smiles`** representation and a **`ccdc.molecule.Molecule.to_string()`** function for returning 'mol2' or 'sdf' formatted string.

```
>>> mol_abebuf.smiles
u'O=C1Nc2ccccc2C(=O)Nc2ccccc12.c1ccncc1'
>>> print mol_abebuf.to_string('mol2')  # doctest: +NORMALIZE_WHITESPACE, +ELLIPSIS
@<TRIPOS>MOLECULE
ABEBUF
...
```

Note that from the SMILES string it is apparent that this molecule contains two disconnected components:

- `O=C1Nc2ccccc2C(=O)Nc2ccccc12`
- `c1ccncc1`

The individual components are molecules as well and can be accessed through the **`ccdc.molecule.Molecule.components`** list and the individual molecule with the greatest molecular weight can be accessed through the **`ccdc.molecule.Molecule.heaviest_component`** attribute.

```
>>> mol_abebuf.components  # doctest: +NORMALIZE_WHITESPACE, +ELLIPSIS
[<ccdc.molecule.Molecule object at ...>,
 <ccdc.molecule.Molecule object at ...>]
>>> [round(mol.molecular_weight, 3) for mol in mol_abebuf.components]
[238.241, 79.1]
>>> mol_abebuf.heaviest_component.smiles
u'O=C1Nc2ccccc2C(=O)Nc2ccccc12'
```

Seealso

The *descriptive molecule documentation* and the **`ccdc.molecule.Molecule`** API documentation.

Crystals

Crystals contain crystallographic descriptors such as Z' (**`ccdc.crystal.Crystal.z_prime`**) and the cell volume (**`ccdc.crystal.Crystal.cell_volume`**).

```
>>> cryst_abebuf.z_prime
1.0
>>> round(cryst_abebuf.cell_volume, 3)  # only want 3 significant figures
3229.582
```

The molecule in the crystal structure can be accessed from the **`ccdc.crystal.Crystal.molecule`** attribute.

```
>>> cryst_abebuf.molecule # doctest: +ELLIPSIS
<ccdc.molecule.Molecule object at ...>
```

```
>>> cryst_abebuf.molecule.smiles
u'O=C1Nc2ccccc2C(=O)Nc2ccccc12.c1ccncc1'
```

Seealso

The *descriptive crystal documentation* and the **ccdc.crystal.Crystal** API documentation.

Entries

CSD entries contains additional information about a crystal structure added during the editorial process. Examples include the chemical name (**ccdc.entry.Entry.chemical_name**) and the publication details (**ccdc.entry.Entry.publication**).

```
>>> entry_abebuf.chemical_name
u'5H,11H-Dibenzo(b,f)(1,5)diazocine-6,12-dione pyridine clathrate'
>>> entry_abebuf.publication  # doctest: +NORMALIZE_WHITESPACE
Citation(authors=u'S.W.Gordon-Wylie, E.Teplin, J.C.Morris,
          M.I.Trombley, S.M.McCarthy, W.M.Cleaver, G.R.Clark',
         journal_name=u'Cryst.Growth Des.',
         volume=u'4', year=2004, first_page=u'789',
         doi=u'10.1021/cg049957u')
```

The crystal and molecule in an entry can be accessed from the **ccdc.entry.Entry.crystal** and **ccdc.entry.Entry.molecule** attributes respectively.

```
>>> entry_abebuf.crystal # doctest: +ELLIPSIS
<ccdc.crystal.Crystal object at ...>
>>> entry_abebuf.crystal.z_prime
1.0
>>> entry_abebuf.molecule # doctest: +ELLIPSIS
<ccdc.molecule.Molecule object at ...>
>>> entry_abebuf.molecule.smiles
u'O=C1Nc2ccccc2C(=O)Nc2ccccc12.c1ccncc1'
```

Reading and writing molecules

Suppose that we wanted to write out our molecule to a file named `abebuf.mol2`.

```
>>> file_name = 'abebuf.mol2'
```

The **ccdc.io** module has three types of writer: **ccdc.io.EntryWriter**, **ccdc.io.CrystalWriter**, **ccdc.io.MoleculeWriter**. These are all context managers that can make use of the Python `with` syntax. This syntax automatically ensures that the writer is closed automatically when the `with` block of code is exited. Let us illustrate this by writing the molecule to the file.

Seealso

For more information on Python's `with` syntax please see PEP 343 www.python.org/dev/peps/pep-0343/.

```
>>> with io.MoleculeWriter(file_name) as mol_writer:
...     mol_writer.write(mol_abebuf)
...
```

Note

In the example above the file format was determined by the file name extension ('mol2'). The file format can also be set explicitly using the `format` argument when instantiating a writer.

Now that we have a molecule written to the file system we can illustrate how to read in a molecule from a file.

```
>>> mol_reader =  io.MoleculeReader(file_name)
>>> mol = mol_reader[0]  # get the first molecule in the file
>>> mol.identifier
u'ABEBUF'
>>> mol.smiles
u'O=C1Nc2ccccc2C(=O)Nc2ccccc12.c1ccncc1'
>>> len(mol.components)
2
```

Note

A **`ccdc.io.MoleculeReader`** is an iterator that can contain zero or more molecules. This is why we had to explicitly get the first molecule from it.

Note also that the concept of a molecule file containing multiple molecules is distinct from the notion that a molecule can contain multiple distinct components. Hence the first molecule extracted from the molecule reader having two components.

Searching the CSD

There are three main ways of searching the CSD: similarity searching (**`ccdc.search.SimilaritySearch`**); text numeric searching (**`ccdc.search.TextNumericSearch`**); and substructure searching (**`ccdc.search.SubstructureSearch`**). The latter can be used to set up searches with 3D geometrical constraints and measurements.

Molecular similarity searching

Suppose that one wanted to find out what spacegroups were occupied by CSD entries that contained compounds similar to the heaviest component of ABEBUF. This type of information may be of interest if one was trying to solve the structure of a new compound using powder data. Having knowledge of which space groups are likely to be occupied by a compound can help the interpretation of the data.

This can be achieved using a **`ccdc.search.SimilaritySearch`**.

```
>>> from ccdc.search import SimilaritySearch
>>> sim_search = SimilaritySearch(mol_abebuf.heaviest_component)
>>> hits = sim_search.search()
>>> len(hits)
23
>>> for h in hits:
...     print '%8s %.3f %s' % (h.identifier, h.similarity, h.crystal.spacegroup_symbol)
...
  ABEBIT 1.000 P-1
  ABEBOZ 1.000 P-1
  ABEBUF 1.000 Pbca
  ISOHAZ 1.000 P-1
  ISOHED 1.000 P212121
  ISOHIH 1.000 P21/n
  LIGREY 0.981 P21/n
```

```
LIGREY01 0.981 P21/c
  DMECDC 0.954 P21
  MEANLD 0.954 P21/a
  UCELIX 0.847 P21/c
  DBZCDC 0.828 P21/a
  ADOHIM 0.806 P21/c
  MANTRN 0.788 P212121
  ASITOL 0.775 Pca21
  ASITUR 0.775 P-1
  ASIVAZ 0.775 P-1
  TOVZEJ 0.759 P-1
  NUJPUE 0.739 P21/c
  TOKZAW 0.739 P-1
  HIRROR 0.735 P21/c
HIRROR01 0.735 P21/n
  JONWOA 0.705 Pbca
```

Seealso

In this example we make use of a `for` loop. If this is unfamiliar to you a basic introduction to the `for` statement can be found in the Python on-line documentation docs.python.org/2/tutorial/controlflow.html#for-statements.

Seealso

The *descriptive similarity search documentation* and the **`ccdc.search.SimilaritySearch`**, **`ccdc.search.SimilaritySearch.SimilarityHit`** API documentation.

Textual and numeric searching

Suppose that one was interested in the physical properties of aspirin and as such wanted to find publications containing aspirin crystal structures with recorded melting points.

This can be achieved using a **`ccdc.search.TextNumericSearch`**.

```
>>> from ccdc.search import TextNumericSearch
>>> text_numeric_search = TextNumericSearch()
>>> text_numeric_search.add_compound_name('aspirin')
>>> identifiers = [h.identifier for h in text_numeric_search.search()]
```

Over the years the assignment of the common and chemical names have somewhat arbitrarily been assigned to the synonyms and compound name fields of the CSD. As such we also need to search the synonyms field for the search to be comprehensive.

```
>>> text_numeric_search.clear()
>>> text_numeric_search.add_synonym('aspirin')
>>> identifiers.extend([h.identifier for h in text_numeric_search.search()])
```

Finally, we loop over the identifiers. To ensure that we avoid duplicates we can make use of Python's built in set functionality. For each identifier we get the relevant entry from the CSD and check if it has a melting point associated with it or not and if it has we write out the identifier, the DOI and the melting point information.

```
>>> for identifier in sorted(set(identifiers)):
...     e = csd_reader.entry(identifier)
...     if e.melting_point:
...         print '%-8s http://dx.doi.org/%-25s %s' % (e.identifier,
```

```
...                                                    e.publication.doi,
...                                                    e.melting_point)
...
ACMEBZ   http://dx.doi.org/10.1107/S0567740881006729 385K
ACSALA13 http://dx.doi.org/10.1021/ja056455b          135.5deg.C
BEHWOA   http://dx.doi.org/10.1107/S0567740882005731 401K
CUASPR01 http://dx.doi.org/10.1107/S1600536803026126 above 573 K
HUPPOX   http://dx.doi.org/10.1039/b208574g           91-96 deg.C
HUPPOX01 http://dx.doi.org/None                       91-96 deg.C
NUWTOP01 http://dx.doi.org/10.1039/c2ce06313a         147 deg.C
```

Seealso

In this example we make use of Python's built-in *set* object. Please see the Python on-line documentation for more information docs.python.org/2/library/stdtypes.html.

Seealso

The *descriptive text numeric search documentation* and the **`ccdc.search.TextNumericSearch`**, **`ccdc.search.TextNumericHit`** API documentation.

Substructure searching with 3D measurements

Identifying identical molecules

Suppose that we wanted to find out if there were any other structures in the CSD that contained `5H,11H-Dibenzo(b,f)(1,5)diazocine-6,12-dione` (the heaviest component of ABEBUF).

This can be achieved by setting up a **`ccdc.search.SubstructureSearch`** with a **`ccdc.search.MoleculeSubstructure`**.

```
>>> from ccdc.search import SubstructureSearch, MoleculeSubstructure
>>> mol_substructure = MoleculeSubstructure(mol_abebuf.heaviest_component)
>>> substructure_search = SubstructureSearch()
>>> sub_id = substructure_search.add_substructure(mol_substructure)
>>> hits = substructure_search.search()
>>> len(hits)
8
>>> for h in hits:
...     print h.identifier
...
ABEBIT
ABEBOZ
ABEBUF
ISOHAZ
ISOHAZ
ISOHED
ISOHIH
ISOHIH
```

Note that some CSD entries were hit more than once by this substructure. If we only want to get one hit per structure we can make use of the `max_hits_per_structure` argument. The example below also illustrates how one can get the indices of the matched atoms out of a substructure hit.

```
>>> hits = substructure_search.search(max_hits_per_structure=1)
>>> len(hits)
6
>>> for h in hits:
...     print h.identifier, h.match_atoms(indices=True)  # doctest: +ELLIPSIS
ABEBIT (26, 27, 0, ..., 23, 21, 11)
ABEBOZ (0, 1, 2, ..., 22, 24, 26)
ABEBUF (0, 1, 2, ..., 25, 26, 27)
ISOHAZ (0, 1, 4, ..., 22, 24, 26)
ISOHED (10, 11, 14, ..., 32, 34, 36)
ISOHIH (10, 11, 14, ..., 32, 34, 36)
```

Torsional preferences of an acyclic fragment

Suppose that we were interested in finding out the torsional preferences of a 2-ethoxyethoxy fragment. Is it 180° like an alkyl chain or not?

This can be achieved by setting up a **`ccdc.search.SubstructureSearch`**, for example from a **`ccdc.search.SMARTSSubstructure`**, and adding a torsion measurement to the search **`ccdc.search.SubstructureSearch.add_torsion_measurement()`**.

First we create a **`ccdc.search.SMARTSSubstructure`** using a SMARTS pattern.

```
>>> from ccdc.search import SMARTSSubstructure
>>> ethoxyethoxy = SMARTSSubstructure('[OD2][CH2]!@[CH2][OD2]')
>>> for qatom in ethoxyethoxy.atoms:  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
...     print qatom.index, qatom
...
0 QueryAtom(O)[elements included: 0: Li ... 101: Rn , equal to 2, atom aromaticity: equal to 0]
1 QueryAtom(C)[hydrogen count, including deuterium: equal to 2, atom aromaticity: equal to 0]
2 QueryAtom(C)[hydrogen count, including deuterium: equal to 2, atom aromaticity: equal to 0]
3 QueryAtom(O)[elements included: 0: Li ... 101: Rn , equal to 2, atom aromaticity: equal to 0]
```

Then we create a **`ccdc.search.SubstructureSearch`** and add the substructure to it.

```
>>> substructure_search = SubstructureSearch()
>>> sub_id = substructure_search.add_substructure(ethoxyethoxy)
```

We can then add a torsion measurement to the substructure using the substructure identifier and the **`ccdc.search.QueryAtom`** indices.

```
>>> substructure_search.add_torsion_angle_measurement('TOR1',
...     sub_id, 0,
...     sub_id, 1,
...     sub_id, 2,
...     sub_id, 3)
```

For the purposes of illustrating this search we will stop the search after the first five structures have been hit.

```
>>> hits = substructure_search.search(max_hit_structures=5)
>>> for h in hits:
...     print '%8s %7.2f' % (h.identifier,
...                          h.measurements['TOR1'])
...
  ABIGEY  176.23
  ABIGEY  162.75
  ABOKAD   64.74
  ABOKAD   58.64
  ABOKAD  -63.53
  ABOKAD   67.31
  ABOKAD   66.00
```

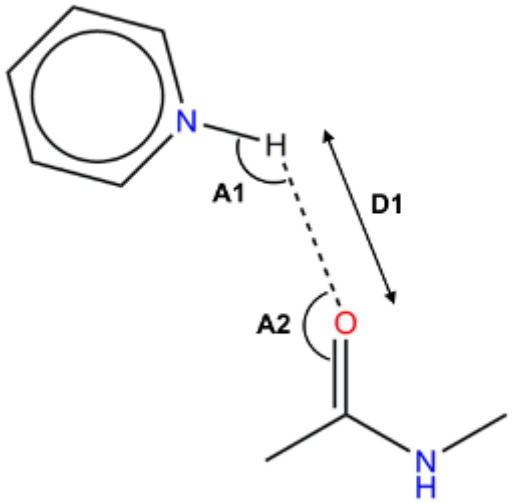
```
  ABOKAD   73.18
  ACIFEZ   70.50
  ACIFEZ   55.07
  ACIFEZ  -70.58
  ACIJOM  180.00
ACIJOM01 -180.00
```

Interaction geometry of a pair of functional groups

Suppose that we were interested in working out if there were any preferences in geometry for a charged pyridine fragment interacting with the carbonyl oxygen of an amide group.

This can be achieved by setting up a **`ccdc.search.SubstructureSearch`**, using for example a pair of **`ccdc.search.SMARTSSubstructure`** instances, and adding a distance constraint between the atoms of interest (**`ccdc.search.SubstructureSearch.add_distance_constraint()`**).

Using the **`ccdc.search.SubstructureSearch.add_angle_measurement()`** one could then set up measurements for the two angles of interest.



Substructure search to investigate the geometric preferences of charged pyridine groups interacting with an amide carbonyl oxygen.

```
>>> charged_pyridine = SMARTSSubstructure('c1ccn(H)cc1')  # indices N:3 H:4
>>> amide = SMARTSSubstructure('C[NH]C(=O)C')  # indices C:2 O:3
>>> substructure_search = SubstructureSearch()
>>> sub1 = substructure_search.add_substructure(charged_pyridine)
>>> sub2 = substructure_search.add_substructure(amide)
```

Note that we have now added two distinct substructures to this particular substructure search. At this point we can set up the distance constraint between the atoms of interest in the two substructures as well as the angle measurements.

```
>>> substructure_search.add_distance_constraint('D1',
...     sub1, 4,
...     sub2, 3,
...     (0, 2.5),  # distance constraint range
...     'Intermolecular')
>>> substructure_search.add_angle_measurement('A1',
...     sub1, 3,
...     sub1, 4,
...     sub2, 3)
```

```
>>> substructure_search.add_angle_measurement('A2',
...     sub2, 2,
...     sub2, 3,
...     sub1, 4)
```

For the purposes of illustrating this search we will stop the search after the first five structures have been hit.

```
>>> hits = substructure_search.search(max_hit_structures=5)
>>> for h in hits:
...     print h.identifier,
...     print '(D1, %.2f)' % h.constraints['D1'],
...     print '(A1, %.2f)' % h.measurements['A1'],
...     print '(A2, %.2f)' % h.measurements['A2']
...
CUWWEX (D1, 1.99) (A1, 157.26) (A2, 142.09)
DOKPAU (D1, 1.93) (A1, 133.90) (A2, 138.39)
DOKPEY (D1, 1.89) (A1, 133.08) (A2, 138.74)
GATLEU (D1, 1.68) (A1, 161.39) (A2, 147.05)
HUKPUY (D1, 2.29) (A1, 118.32) (A2, 98.03)
```

Seealso

The *descriptive substructure search documentation* and the **ccdc.search.SubstructureSearch**, **ccdc.search.MoleculeSubstructure**, **ccdc.search.SMARTSSubstructure**, **ccdc.search.ConnserSubstructure**, **ccdc.search.QuerySubstructure**, **ccdc.search.QueryAtom**, **ccdc.search.QueryBond** API documentation.

Conformational geometry analysis

The CSD-System provides functionality that enables rapid validation of the complete geometry of a given query molecule. This can, for example, be used to validate the binding modes of ligands from protein-ligand structures. It can also provide valuable information in the assessment of the relative stability of conformational polymorphs. In this example we will compare two crystal structures of the HIV-protease inhibitor Ritonavir, identifiers `YIGPIO01` and `YIGPIO02`.

To perform a molecular geometry analysis on a molecule one needs to create a geometry analysis engine.

```
>>> from ccdc import conformer
>>> engine = conformer.GeometryAnalyser()
```

We are only interested in analysing the torsion angles so we will turn off the analysis of bonds, valence angles and rings. In this example we will also turn off search generalisation.

```
>>> engine.settings.bond.analyse = False
>>> engine.settings.angle.analyse = False
>>> engine.settings.ring.analyse = False
>>> engine.settings.generalisation = False
```

To facilitate the analysis we will use a simple function that takes a list of identifiers and prints out the identifier and the number of unusual torsion angles in the structure.

```
>>> def conformation_analysis(identifiers):
...     print 'REFCODE, UnusualTorsions'
...     with io.MoleculeReader('CSD') as reader:
...         for identifer in identifiers:
...             mol = reader.molecule(identifer)
...             checked_mol = engine.analyse_molecule(mol)
```

```
...             num_unusual = len([t for t in checked_mol.analysed_torsions if t.unusual])
...             print "%s, %i" % (identifer, num_unusual)
...
```

Seealso

Here we make use of the concept of a function. If this is unfamiliar to you a basic introduction to functions can be found in the Python on-line documentation docs.python.org/2/tutorial/controlflow.html#defining-functions.

The **ccdc.conformer.GeometryAnalyser.analyse_molecule()** function returns a **ccdc.molecule.Molecule** with dynamically added lists of attributes for features analysed, in this case `analysed_torsions`. These list contain **ccdc.conformer.Analysis** instances, which contains descriptors such as **ccdc.conformer.GeometryAnalyser.Analysis.unusual**. We can use this to work out the number of unusual torsion angles observed in a crystal structures.

```
>>> conformation_analysis(['YIGPIO01', 'YIGPIO02'])
REFCODE, UnusualTorsions
YIGPIO01, 3
YIGPIO02, 0
```

Seealso

The *descriptive molecular geometry analysis documentation* and the **ccdc.conformer** API documentation.

Interaction preference analysis

The CSD-System contains knowledge-based libraries of intermolecular interactions. In this example we will use these libraries to help us understand what interactions would be preferable in a crystal structure of paracetamol (HXACAN).

First of all we create the central and contact group libraries.

```
>>> from ccdc.interaction import InteractionLibrary
>>> central_lib = InteractionLibrary.CentralGroupLibrary()
>>> contact_lib = InteractionLibrary.ContactGroupLibrary()
```

These libraries can be used to identify which central and contact groups are present in the paracetamol molecule.

```
>>> paracetamol = csd_reader.molecule('HXACAN')
>>> central_hits = central_lib.search_molecule(paracetamol)
>>> contact_hits = contact_lib.search_molecule(paracetamol)
```

We can then loop over the central and contact group hits to identify any pairs that have a relative density greater than 1.0.

Note

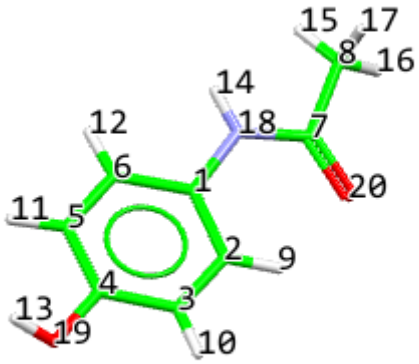
The code below will take a little time to complete.

```
>>> potential_interaction = []
>>> for central_hit in central_hits: # doctest: +SKIP
...     for contact_hit in contact_hits:
...         data = central_hit.group.interaction_data(contact_hit.group)
```

```
...         if data is None:
...             continue
...         rel_density, est_std_dev = data.relative_density
...         if rel_density > 2.0:
...             interaction = (rel_density,
...                            data.ncontacts,
...                            central_hit.name,
...                            [i+1 for i in central_hit.match_atoms(indices=True)],
...                            contact_hit.name,
...                            [i+1 for i in contact_hit.match_atoms(indices=True)])
...             potential_interaction.append(interaction)
...
```

Finally, we sort the interactions by their relative density and print them out.

```
>>> for interaction in sorted(potential_interaction, reverse=True):  # doctest: +ELLIPSIS +SKIP
...     print "%.2f | %i | %s %s | %s %s" % interaction
...
3.82 | 949 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | any OH [19, 13]
3.00 | 112 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | phenol OH [3, 5, 4, 19, 13]
2.97 | 135 | acetylamino [1, 18, 7, 20, 8, 15, 16, 17, 14] | phenol OH [3, 5, 4, 19, 13]
2.79 | 2759 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | any polar X-H (X= N,O or S) [19, 13]
2.79 | 2759 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | any polar X-H (X= N,O or S) [18, 14]
2.54 | 1661 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | any uncharged N-H [18, 14]
2.53 | 1626 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | any uncharged N(sp2)-H [18, 14]
2.50 | 1535 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | uncharged C-NH-C [1, 7, 18, 14]
2.48 | 1429 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | amide N-H [8, 1, 7, 20, 18, 14]
2.41 | 1810 | aromatic-aliphatic amide [2, 6, 1, 8, 18, 7, 20, 14] | any N-H [18, 14]
```



Atom indices of the paracetamol molecule in HXACAN.

Seealso

The *descriptive interaction library documentation* and the `ccdc.interaction` API documentation.

Reading and writing molecules and crystals

Introduction

The `ccdc.io` module is used to read and write molecules and crystals.

```
>>> from ccdc import io
```

Let us also set up a variable for a temporary directory to write files to.

```
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
```

Seealso

API documentation of the io module

Reading molecules

Let us start by finding out what the supported "file" formats for a molecule reader are. The known formats are stored in a dictionary called known_formats in the **`ccdc.io.MoleculeReader`** class.

```
>>> reader_formats = io.MoleculeReader.known_formats.keys()
>>> reader_formats.sort()  # Sort the formats alphabetically
>>> for format in reader_formats:
...     print format
...
aser
cif
identifiers
mol
mol2
res
sdf
sqlite
```

Of the formats above mol2 and sdf are well known chemistry file formats, and mol is a synonym for sdf. The res file format is a well known crystallographic format first introduced by the refinement program SHELX.

Identifiers is a list of molecule identifiers, and can only be used with an accompanying database. In the special case of the CSD this is known as a gcd file. The gcd file format is essentially a list of CSD refcodes and as such is commonly used in the CSD-System to access lists of CSD structures. Below is an example of a valid .gcd file:

```
ACEHAR
ALPLNI
BAXMET
```

The ASER format is the database format of the CSD. It is worth noting that an ASER database is not a single file, it consists of at least three files: an .ind, a .msk and a .tcd file. Furthermore, it can also optionally contain a .inf file.

The distinguished word 'CSD' will open the users' installed CSD database. Note that the **`ccdc.io.MoleculeReader`** will return molecules from the CSD rather than crystals.

```
>>> csd_reader = io.MoleculeReader('CSD')
>>> first_molecule = csd_reader[0]
>>> first_molecule.identifier
u'AABHTZ'
>>> abebuf_mol = csd_reader.molecule('ABEBUF')
>>> abebuf_mol.identifier
u'ABEBUF'
```

Now let us create a molecule file reader from a mol2 file named *pde5_inhibitors.mol2*.

```
>>> filepath = 'pde5_inhibitors.mol2'
```

To get access to the molecules in this file we make use of a **ccdc.io.MoleculeReader**.

```
>>> mol_reader = io.MoleculeReader(filepath)
```

A molecule reader is an iterator from which individual molecules can be accessed by an index.

```
>>> mol = mol_reader[0]
>>> print mol.identifier
1XP0-lig
```

Clearly it is also possible to loop over the molecule reader iterator.

```
>>> for mol in mol_reader:
...     print mol.identifier
...
1XP0-lig
1XOZ-lig
1T9S-lig
```

Note that in the example above the MoleculeReader deduced the file type from the file extension. It is also possible to provide the file type as an optional argument to the MoleculeReader.

Suppose that we had a text file with refcodes.

```
>>> filepath = 'some_refcodes.txt'
```

In order to treat this file as a gcd file when it is read in we make use of the `format` parameter.

```
>>> mol_reader = io.MoleculeReader(filepath, format='identifiers')
>>> for mol in mol_reader:
...     print mol.identifier
ACEHAR
ALPLNI
BAXMET
```

Finally, let us make sure that we have closed the molecule reader.

```
>>> mol_reader.close()
```

Reading crystals

Let us start by finding out what the supported "file" formats for a crystal reader are. The known formats are stored in a dictionary called known_formats in the **ccdc.io.CrystalReader** class.

```
>>> reader_formats = io.CrystalReader.known_formats.keys()
>>> reader_formats.sort()  # Sort the formats alphabetically
>>> for format in reader_formats:
...     print format
...
aser
cif
identifiers
mol
mol2
res
sdf
sqlite
```

These file formats were introduced in the Reading molecules section.

> ***Note***
>
> Although the mol2 file format is usually used for molecules it does support the ability to store crystallographic information using the *@<TRIPOS>CRYSIN* record. The sdf file format, on the other hand, does not support crystallographic information. If a sdf file or a mol2 file (without the *@<TRIPOS>CRYSIN* record) are read in using a **`ccdc.io.CrystalReader`** a default crystal will be created for the molecule (see Default crystal).

Let us create a crystal reader using the *some_refcodes.txt* file in the Reading molecules example.

```
>>> filepath = 'some_refcodes.txt'
```

Again, in order to tell the reader that the input file is in gcd file format we make use of the `format` parameter.

```
>>> crystal_reader = io.CrystalReader(filepath, format='identifiers')
>>> for cryst in crystal_reader:
...     print cryst.spacegroup_symbol
Pbcn
P21/a
P21/a
```

Let us close the crystal reader.

```
>>> crystal_reader.close()
```

Next, let us read crystals from the installed CSD-System.

```
>>> crystal_reader = io.CrystalReader('CSD')
>>> first_crystal = crystal_reader[0]
>>> first_crystal.spacegroup_symbol
u'P-1'
>>> abebuf_crystal = crystal_reader.crystal('ABEBUF')
>>> abebuf_crystal.spacegroup_symbol
u'Pbca'
>>> crystal_reader.close()
```

Finally, let us read crystals from a res file:

```
>>> filepath = 'three_structures.res'
```

Writing molecules

Let us start by finding out what the supported "file" formats for a molecule writer are. The known formats are stored in a dictionary called known_formats in the **`ccdc.io.MoleculeWriter`** class.

```
>>> writer_formats = io.MoleculeWriter.known_formats.keys()
>>> writer_formats.sort()  # Sort the formats alphabetically
>>> for format in writer_formats:
...     print format
...
aser
cif
identifiers
mol
mol2
res
sdf
```

To illustrate how the molecule writer works let us read in molecules from a gcd file and write them out into a sdf file.

```
>>> filepath = 'some_refcodes.txt'
```

In order to tell the reader that the input file is in gcd file format we make use of the `format` parameter.

```
>>> mol_reader = io.MoleculeReader(filepath, format='identifiers')
>>> with io.MoleculeWriter(os.path.join(tempdir, 'some_refcodes.sdf')) as mol_writer:
...     for mol in mol_reader:
...         mol_writer.write(mol)
...
```

> ***Note***
>
> The python `with` syntax automatically ensures that the mol_writer is closed automatically once the `with` block of code is exited. For more information please see PEP 343.

Writing crystals

Let us start by finding out what the supported "file" formats for a crystal writer are. The known formats are stored in a dictionary called known_formats in the **`ccdc.io.CrystalWriter`** class.

```
>>> writer_formats = io.CrystalWriter.known_formats.keys()
>>> writer_formats.sort()  # Sort the formats alphabetically
>>> for format in writer_formats:
...     print format
...
aser
cif
identifiers
mol
mol2
res
sdf
```

To illustrate how the crystal writer works let us read in crystals from a gcd file and write them out into a mol2 file.

```
>>> filepath = 'some_refcodes.txt'
```

In order to tell the reader that the input file is in gcd file format we make use of the `format` parameter.

```
>>> crystal_reader = io.CrystalReader(filepath, format='identifiers')
>>> with io.CrystalWriter(os.path.join(tempdir, 'some_refcodes.cif')) as crystal_writer:
...     for crystal in crystal_reader:
...         crystal_writer.write(crystal)
...
```

Reading Entries

CSD crystallographic database entries can be read from the CSD to gain access to data, such as publication data, not accessible via the crystal. To do this one can create a **`ccdc.io.EntryReader`**:

```
>>> entry_reader = io.EntryReader('CSD')
```

which has all the methods of the other reader classes. For example:

```
>>> first_entry = entry_reader[0]
>>> print first_entry.publication  # doctest: +NORMALIZE_WHITESPACE
```

```
Citation(authors=u'P.-E.Werner', journal_name=u'Cryst.Struct.Commun.',
         volume=u'5', year=1976, first_page=u'873', doi=None)
>>> abebuf_entry = entry_reader.entry('ABEBUF')
>>> print abebuf_entry.publication  # doctest: +NORMALIZE_WHITESPACE
Citation(authors=u'S.W.Gordon-Wylie, E.Teplin, J.C.Morris,
                  M.I.Trombley, S.M.McCarthy, W.M.Cleaver, G.R.Clark',
         journal_name=u'Cryst.Growth Des.', volume=u'4', year=2004,
         first_page=u'789', doi=u'10.1021/cg049957u')
```

EntryReaders also give access to sd-tags for sdf format files and the cif-tags from cif files as strings in a dictionary-like object named `attributes`. This means that one can get access to any property included in a sdf or cif file. To illustrate this let us read in a cif file with many caffeine structures.

```
>>> file_name = 'caffeine.cif'
```

To get access to the raw cif data we need to open the file as an **`ccdc.io.EntryReader`**.

```
>>> reader = io.EntryReader(file_name)
>>> entry_from_cif = reader[0]
>>> entry_from_cif.attributes['_exptl_crystal_colour']
u'orange'
```

Writing Entries

Entries may be written using a **`ccdc.io.EntryWriter`**.

See *Entry documentation* for further details.

Default crystal

When reading in a file without crystallographic information as a crystal or when writing out a molecule without crystallographic information a default crystal will be created.

The default crystal is described in the table below.

Default crystal		
Space group		Unknown
Cell lengths	a	1.0
	b	1.0
	c	1.0
Cell angles	alpha	90.0
	beta	90.0
	gamma	90.0

To illustrate this let us read in a mol2 file without crystallographic information.

```
>>> filepath = '1hak-lig.mol2'
```

To read in the first and only crystal from this file we make use of a **`ccdc.io.CrystalReader`**.

```
>>> crystal_reader = io.CrystalReader(filepath)
>>> crystal = crystal_reader[0]
```

We can now check the default values.

```
>>> crystal.spacegroup_symbol
u'Unknown'
```

```
>>> crystal.cell_lengths
CellLengths(a=1.0, b=1.0, c=1.0)
>>> crystal.cell_angles
CellAngles(alpha=90.0, beta=90.0, gamma=90.0)
```

Working with multiple databases

It is possible to create readers that work with multiple databases simultaneously.

To illustrate this let us create a reader that reads the CSD and all the updates. We start off by identifying the location of the installed CSD database.

```
>>> csd_dir = io.csd_directory()
>>> print csd_dir  # doctest: +SKIP
/home/my/software/csds_v536/csd
```

The CSD and its updates are located in this directory as ASER databases. One of the files used to create the database is the .inf file. We can use this file extension to identify all the databases available.

```
>>> import glob
>>> csd_and_updates = sorted(glob.glob(os.path.join(csd_dir, '*.inf')))
>>> print csd_and_updates  # doctest: +SKIP
[u'/home/my/software/csds_v536/csd/as536be.inf',
 u'/home/my/software/csds_v536/csd/Nov14.inf']
```

Seealso

In the example above we make use of the `glob` module. For more information please see the online Python documentation docs.python.org/2/library/glob.html.

Let us check how many entries are in each database.

```
>>> for db in csd_and_updates:
...     er = io.EntryReader(db)
...     print os.path.basename(db), len(er) # doctest: +SKIP
...
Nov14.inf 26251
as536be.inf 717876
```

Finally, let us create a reader that can work on all the databases.

```
>>> csd_and_updates_reader = io.EntryReader(csd_and_updates)
>>> print len(csd_and_updates_reader) # doctest: +SKIP
744127
```

To work with an in-house ASER database one would use the same process described above; providing a list of paths to .ind files representing the databases of interest.

Working with entries

Introduction

The **`ccdc.entry`** module contains the **`ccdc.entry.Entry`** class which represents a database entry.

Typically a database entry will be read from the CSD. Let us therefore import the **`ccdc.io`** module and read in the first entry from the CSD.

```
>>> from ccdc import io
>>> csd_reader = io.EntryReader('CSD')
>>> first_csd_entry = csd_reader[0]
```

Seealso

API documentation of the entry module

Accessing entry properties

Let us have a look at the crystallographic properties available to us from a CSD crystal structure.

First of all it is worth noting that one can get access to the underlying **`ccdc.crystal.Crystal`** from the entry, and thence to the underlying **`ccdc.molecule.Molecule`**.

```
>>> mol = first_csd_entry.crystal.molecule
>>> mol.identifier
u'AABHTZ'
```

Some entries in the CSD exhibit disorder. By default the **`ccdc.entry.Entry.molecule`** will suppress disordered atoms. For example, 'ABABUB' in the CSD is such an entry:

```
>>> ababub = csd_reader.entry('ABABUB')
>>> mol = ababub.molecule
>>> print len(mol.atoms)
30
>>> print mol.formula
C12 H15 N1 O2
```

A convention within the CSD is that disordered atoms have labels ending with a '?'. The full molecule, with suppressed atoms, can be retrieved:

```
>>> disordered_mol = ababub.disordered_molecule
>>> print len(disordered_mol.atoms)
42
>>> print disordered_mol.formula
C16 H23 N1 O2
```

For entries that have been accessed from the CSD-System one can also access the chemical name and formula of the crystal content.

```
>>> print first_csd_entry.chemical_name
4-Acetoamido-3-(1-acetyl-2-(2,6-dichlorobenzylidene)hydrazine)-1,2,4-triazole
>>> print first_csd_entry.formula
C13 H12 Cl2 N6 O2
```

Let us illustrate some more properties available for CSD entries using a crystal structure of Ibuprofen.

```
>>> ibuprofen = csd_reader.entry('IBPRAC')
>>> print ibuprofen.has_3d_structure
True
>>> print ibuprofen.has_disorder
False
>>> print ibuprofen.is_organometallic
False
>>> print ibuprofen.is_polymeric
False
```

```
>>> print ibuprofen.bioactivity
analgesic and antiinflammatory agent
>>> print ibuprofen.synonyms
(u'Ibuprofen', u'Advil', u'Motrin', u'Nurofen')
```

This particular CSD entry does not have a DOI.

```
>>> print ibuprofen.publication.doi
None
```

However where it is defined it will be returned.

```
>>> print csd_reader.entry('ABEBUF').publication.doi
10.1021/cg049957u
>>> print csd_reader.entry('ABEBUF').publication  # doctest: +NORMALIZE_WHITESPACE
Citation(authors=u'S.W.Gordon-Wylie, E.Teplin, J.C.Morris, M.I.Trombley,
                  S.M.McCarthy, W.M.Cleaver, G.R.Clark',
         journal_name=u'Cryst.Growth Des.', volume=u'4', year=2004,
         first_page=u'789', doi=u'10.1021/cg049957u')
```

A publication is returned as a named tuple, whose members may be retrieved by name or index, containing authors, journal_name, volume, year, number of first page and the publication's doi where present.

```
>>> print ibuprofen.publication  # doctest: +NORMALIZE_WHITESPACE
Citation(authors=u'J.F.McConnell',
         journal_name=u'Cryst.Struct.Commun.', volume=u'3', year=1974,
         first_page=u'73', doi=None)
>>> print ibuprofen.publication.authors
J.F.McConnell
```

From a CSD entry it is also possible to get hold of information on the colour, melting point, polymorphic form description, any disorder and the radiation source of the crystal's determination when this information is available in the underlying CSD entry.

```
>>> print ibuprofen.color
None
>>> print ibuprofen.melting_point
None
>>> print ibuprofen.polymorph
polymorph 1
>>> print ibuprofen.disorder_details
None
>>> print csd_reader.entry('ABABUB').disorder_details
The cyclohexene ring is disordered over two sites with occupancies 0.5878:0.4122.
>>> print csd_reader.entry('ABINOR01').radiation_source
Neutron
```

The entry for ibuprofen has no editorial comments; where they are present they may represent editorial decisions made during structural curation, or patent information:

```
>>> print ibuprofen.remarks
None
>>> print csd_reader.entry('ABAPCU').remarks
The position of the hydrate is dubious. It has been deleted
>>> print csd_reader.entry('ARISOK').remarks
U.S. Patent: US 6858644 B2
```

An indication of whether an experiment was done at pressure is available. This is given as a string and the units have not yet been normalised. Where this is None, the experiment was performed at ambient pressure:

```
>>> print ibuprofen.pressure
None
>>> print csd_reader.entry('ABULIT03').pressure
at 1.4 GPa
```

An boolean property will indicate whether or not the crystallographic determination was performed in a powder study:

```
>>> print ibuprofen.is_powder_study
False
>>> print csd_reader.entry('ACATAA').is_powder_study
True
```

Within the CSD many structures are cross-referenced. These can be obtained and inspected as follows:

```
>>> cross_refs = ibuprofen.cross_references
>>> print cross_refs
(CrossReference(for stereoisomer see [JEKNOC]),)
>>> xref = cross_refs[0]
>>> print xref.text
for stereoisomer see [JEKNOC]
>>> print xref.type
Stereoisomer
>>> print xref.scope
Family
>>> print xref.identifiers
('JEKNOC',)
```

These properties would not be available for an entry read in from, for example, a mol2 file.

```
>>> filepath = 'ABEBUF.mol2'
```

To get access to the entry in this file we make use of a **ccdc.io.EntryReader**.

```
>>> entry_reader = io.EntryReader(filepath)
>>> entry_from_mol2 = entry_reader[0]
>>> entry_reader.close()
>>> print entry_from_mol2.chemical_name
None
>>> print entry_from_mol2.formula
C19 H15 N3 O2
```

Where the database comes from an sdf file the EntryReader will give access to the SDF tags for the entry via the entry.attributes property:

```
>>> file_name = 'gold_output.sdf'
```

To get access to the entry in this file we make use of a **ccdc.io.EntryReader**.

```
>>> reader = io.EntryReader(file_name)
>>> entry_from_sdf = reader[0]
>>> for k, v in entry_from_sdf.attributes.items():
...     print k
...     print v
...
Gold.Chemscore.Rot
1.7155
Gold.Protein.RotatedAtoms
   42.8596   40.6557   11.7180 H   0  0  0  0  0  0  0  0  0  0  0  0  # atno 6359 bound_to 3214
   39.9356   46.2719   13.3012 H   0  0  0  0  0  0  0  0  0  0  0  0  # atno 6443 bound_to 742
```

```
Gold.Chemscore.DEClash.Weighted
17.1265
Gold.Chemscore.Lipo.Weighted
-30.1150
Gold.Chemscore.Rot.Weighted
4.3916
Gold.Chemscore.Hbond.Weighted
-3.3317
Gold.Chemscore.ZeroCoef
-5.4800
Gold.Chemscore.Lipo
257.3928
Gold.Chemscore.DEInternal.Weighted
2.1334
Gold.Chemscore.Internal_Hbond.Weighted
0.0000
Gold.Chemscore.Hbond
0.9975
Gold.Chemscore.DEClash
17.1265
Gold.Chemscore.DEInternal
2.1334
Gold.Chemscore.DG
-34.5350
Gold.Protein.ActiveResidues
 PHE87   TYR96   PHE98   THR101  MET184  THR185  LEU244  VAL247  GLY248  THR252
 VAL295  ASP297  ILE395  VAL396  HEM1
Gold.Chemscore.Internal_Hbond
0.0000
Gold.Chemscore.Metal.Weighted
0.0000
Gold.Chemscore.Metal
0.0000
Gold.Chemscore.Fitness
15.2752
>>> reader.close()
```

The values of the attributes dict are all strings: it is left to the user to convert as appropriate.

An entry may be constructed from a molecule with attributes constructed from arbitrary keyword parameters.

```
>>> from ccdc.entry import Entry
>>> aabhtz_entry = Entry.from_molecule(mol, annotation='First structure in CSD')
```

Attributes may also be added to an entry after it has been instantiated.

```
>>> aabhtz_entry.attributes['molecular_weight'] = mol.molecular_weight
```

Note that attributes of any value may be added to the entry. These attributes will be written to an sdf format file if an EntryWriter is used:

```
>>> with io.EntryWriter('aabhtz.sdf') as writer:
...     writer.write(aabhtz_entry)
```

Working with crystals

Introduction

The **ccdc.crystal** module contains the **ccdc.crystal.Crystal** class.

However, you are unlikely to use this class to create a crystal directly. You are much more likely to come across a crystal object after having read in a crystal from a file. Let us therefore import the **`ccdc.io`** module and read in the first crystal structure from the CSD.

```
>>> from ccdc import io
>>> csd_reader = io.CrystalReader('CSD')
>>> first_csd_entry = csd_reader[0]
```

Seealso

API documentation of the crystal module

Accessing crystallographic properties

Let us have a look at the crystallographic properties available to us from a CSD crystal structure.

First of all it is worth noting that one can get access to the underlying **`ccdc.molecule.Molecule`** from the crystal.

```
>>> mol = first_csd_entry.molecule
>>> mol.identifier
u'AABHTZ'
```

Warning

If the crystal structure has no 3D information about the molecule's atoms accessing the **`ccdc.crystal.Crystal.molecule`** will throw a `TypeError`. If one has access to the corresponding **`ccdc.entry.Entry`** the molecule can be constructed from that.

Some crystals read from the CSD will contain disorder. In this case disordered atoms will be suppressed and not present in the return molecule. If required the full molecule, complete with disordered atoms may be retrieved:

```
>>> c = csd_reader.crystal('ABABUB')
>>> mol = c.molecule
>>> print len(mol.atoms)
30
>>> dis = c.disordered_molecule
>>> print len(dis.atoms)
42
```

The basic descriptors of a crystal structure are its cell lengths, cell angles and its space group.

```
>>> print "a: %.2f b: %.2f c: %.2f" % first_csd_entry.cell_lengths
a: 11.37 b: 10.27 c: 7.36
>>> print "alpha: %.2f beta: %.2f gamma: %.2f" % first_csd_entry.cell_angles
alpha: 108.75 beta: 71.07 gamma: 96.16
>>> print first_csd_entry.spacegroup_symbol
P-1
>>> print first_csd_entry.symmetry_operators
(u'x,y,z', u'-x,-y,-z')
>>> print first_csd_entry.symmetry_rotation('-x,-y,-z')
(-1, 0, 0, 0, -1, 0, 0, 0, -1)
>>> print first_csd_entry.symmetry_translation('-x,-y,-z')
(0.0, 0.0, 0.0)
```

> ***Note***
>
> The example above makes use of Python's string formatting operator. For more information please see the Python string formatting documentation.

Other, basic crystallographic properties include the Z prime, the Z value the cell volume and the calculated density.

```
>>> first_csd_entry.z_prime
1.0
>>> first_csd_entry.z_value
2.0
>>> round(first_csd_entry.cell_volume, 2)
769.98
>>> round(first_csd_entry.calculated_density, 2)
1.53
```

> ***Note***
>
> The example above makes use of Python's built in round function. For more information please see the Python documentation.

For crystals that have been accessed from the CSD-System one can also access the chemical name and formula of the crystal content.

```
>>> print first_csd_entry.formula
C13 H12 Cl2 N6 O2
```

Let us illustrate that these would not be available for a crystal read in from, for example, a mol2 file.

```
>>> filepath = 'ABEBUF.mol2'
```

To get access to the crystal in this file we make use of a **`ccdc.io.CrystalReader`**.

```
>>> cryst_reader = io.CrystalReader(filepath)
>>> cryst_from_mol2 = cryst_reader[0]
>>> cryst_reader.close()
>>> print cryst_from_mol2.formula
C19 H15 N3 O2
```

A **`ccdc.io.CrystalReader`** also provides access to attributes such as cell volume, lengths and angles.

```
>>> round(cryst_from_mol2.cell_volume, 2)
3229.58
>>> print "a: %.2f b: %.2f c: %.2f" % cryst_from_mol2.cell_lengths
a: 10.72 b: 10.90 c: 27.64
>>> print "alpha: %.2f beta: %.2f gamma: %.2f" % cryst_from_mol2.cell_angles
alpha: 90.00 beta: 90.00 gamma: 90.00
>>> print cryst_from_mol2.spacegroup_symbol
Pbca
```

The mol2 file format specification for crystals does not include any information about the Z' and Z value. These attributes consequently are set to *None*.

```
>>> print cryst_from_mol2.z_prime
None
```

```
>>> print cryst_from_mol2.z_value
None
```

Manipulating crystals

The **ccdc.crystal.Crystal.molecular_shell()** function can be used to generate a packing shell around the molecules in a crystal or a subset of atoms of the molecules in the crystal.

Seealso

The API documentation **ccdc.crystal.Crystal.molecular_shell()** and the "Generating molecular shells" example in the *cookbook documentation*.

Working with molecules, atoms and bonds

Introduction

The classes **ccdc.molecule.Atom**, **ccdc.molecule.Bond** and **ccdc.molecule.Molecule** are all stored in the **ccdc.molecule** module.

However, you are unlikely to use these classes to create objects directly. You are much more likely to come across them after having read in a molecule from a file or after having accessed one from a database. Let us therefore import the **ccdc.io** module to allow us to get access to some molecules.

```
>>> from ccdc import io
>>> from ccdc.molecule import Molecule
```

Seealso

API documentation of the molecule module

A ccdc.molecule.Molecule can contain several molecules

What - a molecule can contain several molecules??? Yes, in the CSD Python API we refer to these sub-molecules as components.

The reason why this arises is that most molecular file formats allow the inclusion of molecular species which are not covalently bonded to each other to be included in the same entry. Below for example is a SMILES representation of the CSD structure ABEBUF.

```
O=C1Nc2ccccc2C(=O)Nc2ccccc12.c1ccncc1
```

This "molecule" contains two components 5H,11H-Dibenzo(b,f)(1,5)diazocine-6,12-dione and pyridine.

Note

It is similarly possible to represent several molecules within single entries of mol2 and sdf files. In mol2 file format these are referred to as substructures.

Let us illustrate this using the CSD structure ABEBUF.

```
>>> filepath = 'ABEBUF.mol2'
```

To get access to the molecule in this file we make use of a **ccdc.io.MoleculeReader**.

```
>>> mol_reader = io.MoleculeReader(filepath)
>>> mol = mol_reader[0]
>>> print mol.identifier
ABEBUF
>>> print mol.smiles
O=C1Nc2ccccc2C(=O)Nc2ccccc12.c1ccncc1
>>> len(mol.components)
2
>>> for component in mol.components:
...     print component.smiles
...
O=C1Nc2ccccc2C(=O)Nc2ccccc12
c1ccncc1
```

This has implications for molecular properties, which are calculated using all the components within a molecule. Let us illustrate this using the molecular weight.

```
>>> print mol.molecular_weight
317.3407
>>> for component in mol.components:
...     print component.molecular_weight
...
238.241
79.0997
```

Finally, it is worth noting that in many cases when dealing with molecules containing salts and solvents one is not particularly interested in the salts or solvents *per se*, but rather the larger organic molecule in the structure. This "heaviest" component can be accessed as illustrated below.

```
>>> heaviest_component = mol.heaviest_component
>>> print heaviest_component.identifier, heaviest_component.molecular_weight
01 238.241
```

Standardising and editing molecules

There are some functions available for standardising and editing molecules. Let us start by illustrating how one can remove and add hydrogen atoms.

```
>>> print mol.molecular_weight
317.3407
>>> mol.remove_hydrogens()
>>> print mol.molecular_weight
302.2222
>>> mol.add_hydrogens()
>>> print mol.molecular_weight
317.3407
```

To illustrate some of the more advanced molecule standardisation functions let us read in a PDB ligand (from the structure 1f0r), which has not got any hydrogen atoms added to it, in sdf file format.

```
>>> filepath = '1f0r-lig.sdf'
```

To get access to the molecule in this file we make use of a **`ccdc.io.MoleculeReader`**.

```
>>> mol_reader = io.MoleculeReader(filepath)
>>> mol = mol_reader[0]
>>> print mol.identifier
1f0r-lig
```

```
>>> print mol.molecular_weight
434.3864
```

First of all if I am unsure about the bond typing I will need to assign this first. Note that this bond typing is based on the geometry of the molecule.

```
>>> mol.assign_bond_types()
```

By default, the **ccdc.molecule.Molecule.assign_bond_types()** function guesses the bond types of all bonds. However, if you had a molecule that only had the bond types set for some of the bonds and you had confidence in these then you could guess the bond types of only those bonds with "unknown" types by setting the parameter *which* to *unknown*.

If you prefer to have alternating single and double bonds rather than aromatic bonds you can call:

```
>>> mol.kekulize()
```

After the bond types have been assigned one can add hydrogen atoms to the molecule.

```
>>> mol.add_hydrogens()
```

Let us check if the molecule has had its molecular weight increased

```
>>> print mol.molecular_weight
453.5365
```

If you had a molecule with bond types that you had confidence in, but you wanted to standardise the aromatic and delocalised bonds CSD conventions then you could make use of the functions:

- **ccdc.molecule.Molecule.standardise_aromatic_bonds()**
- **ccdc.molecule.Molecule.standardise_delocalised_bonds()**

Finally, let us come back to the concept of a molecule having multiple components. Suppose that we wanted to add the heaviest component from the ABEBUF molecule to the 1f0r ligand. This could be achieved with the **ccdc.molecule.Molecule.add_molecule()** function.

```
>>> mol.add_molecule(heaviest_component)
>>> for component in mol.components:
...     print component.identifier
...
01
02
```

> ***Note***
>
> Operations which change the underlying molecule of an entry or a crystal, such as adding hydrogens, or assigning bond types will permanently affect the molecule. Any subsequent access of the molecule of an entry or crystal will return the mutated molecule. Reading the entry or crystal afresh from a database will, of course, return the original, unmutated molecule.

Accessing molecular properties

Let us now have a look at the molecular properties available to us using the ABEBUF structure.

```
>>> filepath = 'ABEBUF.mol2'
```

To get access to the molecule in this file we make use of a **ccdc.io.MoleculeReader**.

```
>>> mol_reader = io.MoleculeReader(filepath)
>>> mol = mol_reader[0]
```

Lists of the atoms and bonds in a molecule are accessed using the attributes with these names.

```
>>> mol.atoms  # doctest: +ELLIPSIS
[Atom(O1), Atom(O2), Atom(N1), ... Atom(H13), Atom(H14), Atom(H15)]
>>> len(mol.atoms)
39
>>> mol.heavy_atoms # doctest: +ELLIPSIS
[Atom(O1), Atom(O2), ... Atom(C17), Atom(C18), Atom(C19)]
>>> mol.bonds  # doctest: +ELLIPSIS
[Bond(Double Atom(O1) Atom(C1)), ... Bond(Single Atom(H15) Atom(C19))]
>>> len(mol.bonds)
41
```

we can gain access to particular atoms and bonds of the molecule if we know their index in the lists. Alternatively, we can gain access to particular atoms and bonds of the molecule by atom labels, provided the labels used are unique within the molecule:

```
>>> mol.atom('O2')
Atom(O2)
>>> mol.bond('C1', 'O1')
Bond(Double Atom(O1) Atom(C1))
```

The atoms and bonds have a Sybyl type:

```
>>> mol.atom('N1').sybyl_type
u'N.am'
>>> mol.bond('N1', 'C1').sybyl_type
u'am'
```

An atom has a Van der Waals radius:

```
>>> mol.atom('O1').vdw_radius
1.52
>>> mol.atom('C1').vdw_radius
1.7
```

We can also access the formal charge of the molecule as well as the molecular weight and the molecular formula.

```
>>> mol.formal_charge
0
>>> mol.molecular_weight
317.3406999999999
>>> mol.formula
'C19 H15 N3 O2'
```

Note that the molecular formula is the sum of all the components in the molecule. To get a component-wise formula one can for example:

```
>>> '.'.join(c.formula for c in mol.components)
'C14 H10 N2 O2.C5 H5 N1'
```

There is also a property which tells you whether or not the molecule is organic (as opposed to metal-organic).

```
>>> mol.is_organic
True
```

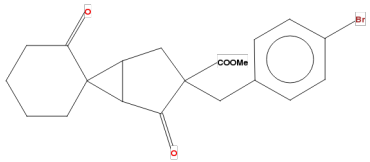
It is also worth mentioning that the molecule identifier and SMILES representation can be accessed through the molecule.

```
>>> mol.identifier
u'ABEBUF'
>>> mol.smiles
u'O=C1Nc2ccccc2C(=O)Nc2ccccc12.c1ccncc1'
```

Rings

We can get access to the rings of a molecule. For example, let us read in 'CEMGUY' from the CSD which has an interesting set of rings:

```
>>> csd = io.MoleculeReader('CSD')
>>> cemguy = csd.molecule('CEMGUY')
```



CEMGUY 2D diagram, to exemplify ring analysis.

We can immediately get access to the size of the largest and smallest ring of this molecule:

```
>>> print cemguy.largest_ring_size, cemguy.smallest_ring_size
6 3
```

The molecule has a property which will give access to the basic rings of the structure. The set of rings returned will be all simple rings, not an enclosing, larger ring. In this example, there will be a 3-membered ring, a 5-membered ring and two six-membered rings, but not the six-membered ring containing the 3- and 5-membered rings:

```
>>> rings = cemguy.rings
>>> print len(rings)
4
>>> print([len(r) for r in rings])
[3, 5, 6, 6]
```

We can get the atoms and bonds of each ring:

```
>>> print rings[0].atoms # doctest:+ELLIPSIS
[Atom(...), Atom(...), Atom(...)]
>>> print rings[0].bonds # doctest:+ELLIPSIS
[Bond(Single Atom(...) Atom(...)), Bond(Single Atom(...) Atom(...)), Bond(Single Atom(...) Atom(...))]
```

Note that the atoms of the ring are ordered such that there is a bond between consecutive atoms. We can test for containment in a ring:

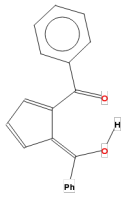
```
>>> print cemguy.atom('C1') in rings[0]
True
>>> print cemguy.bond('C6', 'C1') in rings[0]
True
```

There are a couple of predicates on rings:

```
>>> print rings[0].is_aromatic
False
>>> print rings[-1].is_aromatic
```

```
True
>>> print rings[0].is_fused
True
>>> print rings[-1].is_fused
False
>>> print rings[0].is_fully_conjugated
False
>>> print rings[-1].is_fully_conjugated
True
```

The predicate 'is_fully_conjugated' is true for aromatic ring systems and for ring systems where every single bond is conjugated, for example the 5-membered ring in 'POCTOS'.



POCTOS 2D diagram to exemplify conjugated rings.

```
>>> poctos = csd.molecule('POCTOS').components[0]
>>> poctos_rings = poctos.rings
>>> print poctos_rings[0].is_aromatic
False
>>> print poctos.rings[0].is_fully_conjugated
True
```

The atom 'C6' in 'CEMGUY' is a spiro atom and is in two rings; the atom 'C5' is not spiro but is also in two rings. The Atom property, rings, gives access to the list of all rings in which the atom lies, and the Bond property, rings, similarly gives access to the rings of which the bond forms part:

```
>>> print cemguy.atom('C6').is_spiro
True
>>> print len(cemguy.atom('C6').rings)
2
>>> print cemguy.atom('C5').is_spiro
False
>>> print cemguy.atom('C5').rings # doctest:+ELLIPSIS
[Atom(...)-Atom(...)-Atom(...), Atom(...)-Atom(...)-Atom(...)-Atom(...)-Atom(...)]
>>> print cemguy.atom('C12').rings
[]
>>> print len(cemguy.bond('C1', 'C5').rings)
2
```

Some of the bonds of 'POCTOS' are conjugated:

```
>>> poctos.bond('C6', 'C7').is_conjugated
True
>>> poctos.bond('C1', 'O1').is_conjugated
False
```

We are not analysing the geometry of the bonded system in order to determine conjugation: this may produce false positives.

Working with string representations of mol2 and sdf files

It is sometimes useful to have access to a string representation of a molecule within a Python script, for example to convert a CCDC molecule into an RDKit molecule by making use of a SDF formatted string as an intermediate.

The **`ccdc.molecule.Molecule.to_string()`** method is therefore provided for a molecule to be returned as a string in either 'mol2' or 'sdf' file format.

```
>>> s = mol.to_string(format='mol2')
>>> print s # doctest: +NORMALIZE_WHITESPACE, +ELLIPSIS
@<TRIPOS>MOLECULE
ABEBUF
    39    41     2     0     0
SMALL
NO_CHARGES
****
Generated from the CSD
<BLANKLINE>
@<TRIPOS>ATOM
     1 O1        1.9287    5.1676    1.7209    O.2        1 RES1    0.0000
...
```

Similarly the **`ccdc.molecule.Molecule.from_string()`** method reads in a molecule from a 'mol2', 'sdf', 'mol', or 'cif' formatted string.

```
>>> m = Molecule.from_string(s, format='mol2')
>>> print m.identifier, m.molecular_weight
ABEBUF 317.3407
```

The format parameter may be one of 'mol2', 'sdf', 'mol', or 'cif'.

> ***Note***
>
> A SMILES representation of the molecule is accessible through its **`ccdc.molecule.Molecule.smiles`** attribute.

Accessing atomic properties

Let us have a look at the atomic properties available to us using the ABEBUF structure. To do this we will need an atom. Let us just use the first one in the molecule.

```
>>> atom = mol.atoms[0]
```

We can identify the atom using the index and/or the label.

```
>>> atom.index
0
>>> atom.label
u'O1'
```

To understand the chemistry we can inspect the atomic symbol and the formal charge.

```
>>> atom.atomic_symbol
u'O'
>>> atom.formal_charge
0
```

Furthermore, there are also descriptors to check whether or not an atom is a metal, a hydrogen bond donor, a hydrogen bond acceptor and whether or not it is in a ring system.

```
>>> atom.is_metal
False
>>> atom.is_donor
False
>>> atom.is_acceptor
True
>>> atom.is_cyclic
False
```

Clearly, we also have access to the atomic coordinates.

```
>>> atom.coordinates
Coordinates(x=1.9287, y=5.1676, z=1.7209)
>>> [round(a, 4) for a in atom.fractional_coordinates]
[0.1799, 0.4741, 0.0623]
```

> ***Note***
>
> The coordinates returned by **`ccdc.molecule.Atom.coordinates()`** is in orthogonal space. If the atom has no coordinates, None will be returned. Similarly fractional_coordinates will return None if no coordinate information is available.

There is a convenience property to determine whether all the heavy atoms of a molecule have 3D coordinate information:

```
>>> mol.is_3d
True
```

We have access to the neighbouring atoms:

```
>>> atom.neighbours
(Atom(C1),)
```

and to the bonds to this atom:

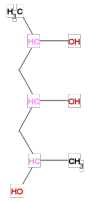
```
>>> atom.bonds
(Bond(Double Atom(O1) Atom(C1)),)
```

We can access R/S chirality information for atoms:

```
>>> duxyea = io.MoleculeReader('CSD').molecule('DUXYEA')
>>> print [(a.label, a.chirality) for a in duxyea.atoms if a.is_chiral] # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
[(u'C1', 'S'), (u'C3', 'R'), (u'C6', 'R'), (u'C7', 'S')...]
```

We will also detect para-chiral centres where the chirality of the atom depends only on the chirality of connected atoms:

```
>>> hptrol = io.MoleculeReader('csd').molecule('HPTROL')
>>> hptrol.add_hydrogens()
>>> print [(a.label, a.chirality) for a in hptrol.atoms if a.is_chiral] # doctest: +NORMALIZE_WHITESPACE
[(u'C2', 'R'), (u'C4', 'S'), (u'C6', 'S')]
```



HPTR0L 2D diagram, to exemplify para-chiral centres.

Here the central C4 atom is para-chiral since the atoms C2 and C6 have opposite chirality.

Accessing bond properties

Let us have a look at the bond properties available to us using the ABEBUF structure. To do this we will need a bond. Let us just use the first one in the molecule.

```
>>> bond = mol.bonds[0]
```

To identify the bond we can access the two atoms connected by it.

```
>>> atom1, atom2 = bond.atoms
>>> print atom1.label, atom2.label
O1 C1
```

To understand the chemistry we can check the bond type.

```
>>> print bond.bond_type
Double
```

Note that we can also use integers to check the type of a bond.

```
>>> bond.bond_type == 1
False
>>> bond.bond_type == 2
True
```

The integers used to represent bond types according to the CSD convention are.

Type	Integer
Unknown	0
Single	1
Double	2
Triple	3
Quadruple	4
Aromatic	5
Delocalised	7
Pi	9

It is also possible to check whether or not a bond is cyclic and/or rotatable.

```
>>> bond.is_cyclic
False
>>> bond.is_rotatable
False
```

Editing molecules

Introduction

The classes **ccdc.molecule.Atom**, **ccdc.molecule.Bond** and **ccdc.molecule.Molecule** all support editing of their attributes. This enables both *de novo* creation of new molecules and modification of existing molecules. Let us firstly exemplify this by changing 'AABHTZ' into something unrecognisable.

We must import the **ccdc.io** module to get access to the CSD, and read in 'AABHTZ':

```
>>> from ccdc import io
>>> csd = io.MoleculeReader('csd')
```

```
>>> aabhtz = csd.molecule('AABHTZ')
```

Atom editing

Now we can change atom types, for example change the chlorine atoms to another halogen:

```
>>> for a in aabhtz.atoms:
...     if a.atomic_symbol == 'Cl':
...         a.atomic_symbol = 'Br'
```

Equivalently we could have changed the atoms' atomic_number:

```
>>> for a in aabhtz.atoms:
...     if a.atomic_number == 17:
...         a.atomic_number = 35
```

These atoms now have inappropriate labels:

```
>>> print [a.label for a in aabhtz.atoms if a.atomic_symbol == 'Br']
[u'Cl1', u'Cl2']
```

which we can change:

```
>>> for i, a in enumerate(aabhtz.atoms):
...     if a.atomic_symbol == 'Br':
...         a.label = 'Br%d' % (i+1)
>>> print [a.label for a in aabhtz.atoms if a.atomic_symbol == 'Br']
[u'Br1', u'Br2']
```

Other attributes of the atoms can be changed, for example, **ccdc.molecule.Atom.formal_charge**, **ccdc.molecule.Atom.coordinates** for those who wish to set the formal charges and coordinates, respectively, of atoms explicitly.

Bond editing

The type of a bond may be changed, using either a string representation, an integer or a **ccdc.molecule.Bond.BondType**. Let us change the double bonded oxygens to acids:

```
>>> for b in aabhtz.bonds:
...     if 'O' in [a.atomic_symbol for a in b.atoms]:
...         b.bond_type = 'Single'
```

It is important to now make sure hydrogens are added appropriately:

```
>>> aabhtz.add_hydrogens()
```

Molecule editing

Molecules may be constructed from scratch, or an existing molecule edited.

Building molecules from scratch

It is possible to build up a molecule from scratch. To illustrate how to construct a molecule let us create a charged pyridine fragment.

First of all let us import the required classes from the **`ccdc.molecule`** module.

```
>>> from ccdc.molecule import Molecule, Atom, Bond
```

Now we need to create a **`ccdc.molecule.Molecule`** instance.

```
>>> mol = Molecule(identifier='my molecule')
```

Then we create the **`ccdc.molecule.Atom`** instances.

```
>>> a1 = Atom('N', coordinates=(-0.301, -0.968, 4.080), formal_charge=1)
>>> a2 = Atom('C', coordinates=(-1.590, -1.256, 4.148))
>>> a3 = Atom('C', coordinates=(-2.144, -2.420, 3.669))
>>> a4 = Atom('C', coordinates=(-1.327, -3.345, 3.075))
>>> a5 = Atom('C', coordinates=( 0.200, -3.055, 2.977))
>>> a6 = Atom('C', coordinates=( 0.447, -1.874, 3.505))
```

Once created the **`ccdc.molecule.Atom`** instances can be added to the molecule.

```
>>> a1_id = mol.add_atom(a1)
>>> a2_id = mol.add_atom(a2)
>>> a3_id = mol.add_atom(a3)
>>> a4_id = mol.add_atom(a4)
>>> a5_id = mol.add_atom(a5)
>>> a6_id = mol.add_atom(a6)
```

The bonds can then be added between the atoms.

```
>>> aromatic_bond_type = Bond.BondType(5)
>>> b1_id = mol.add_bond(aromatic_bond_type, a1_id, a2_id)
>>> b2_id = mol.add_bond(aromatic_bond_type, a2_id, a3_id)
>>> b3_id = mol.add_bond(aromatic_bond_type, a3_id, a4_id)
>>> b4_id = mol.add_bond(aromatic_bond_type, a5_id, a6_id)
>>> b5_id = mol.add_bond(aromatic_bond_type, a6_id, a1_id)
```

Finally, let us add hydrogen atoms to the molecule.

```
>>> mol.add_hydrogens()
```

Let us view our creation in mol2 file format.

```
>>> print mol.to_string('mol2')  # doctest: +NORMALIZE_WHITESPACE, +ELLIPSIS
@<TRIPOS>MOLECULE
my molecule
    14    13     1     0     0
SMALL
USER_CHARGES
****
Generated from the CSD
...
```

However there are easier ways to construct molecules using geometries from the CSD. We can construct the same molecule by taking 'ABADUE' from the CSD, which contains a pyridine, and removing the remaining atoms.

```
>>> abadue = csd.molecule('ABADUE')
```

We could remove the other atoms like this:

```
>>> keep_atom_labels = ['C1', 'C2', 'C3', 'C4', 'C5', 'N1', 'H1', 'H2', 'H3', 'H4']
>>> for a in abadue.atoms:
...     if a.label not in keep_atom_labels:
...         abadue.remove_atom(a)
```

or more succinctly by a generator expression:

```
>>> abadue.remove_atoms(
...     a for a in abadue.atoms if a.label not in keep_atom_labels
... )
```

But in this instance the atoms to remove form a connected group of atoms downstream of the bond between atoms C5 and C6. There is a single API call to achieve this:

```
>>> abadue = csd.molecule('ABADUE')
>>> c5 = abadue.atom('C5')
>>> c6 = abadue.atom('C6')
>>> abadue.remove_group(c5, c6)
```

There are analogous methods to remove a bond, or to remove a set of bonds, which may be used to break rings or to disconnect a molecule into separate components.

We can substitute groups, in a similar way to that of the **`ccdc.molecule.Molecule.remove_group()`** above. In this case we will preserve the geometry of the substituting group, by translating and rotating it so that the the orientation of the substituted group is preserved.

For example 'ABFMPB' has a benzamide group, and we might wish to substitute the benzene by a smaller lipophilic moiety, perhaps to fit better into a binding site. The same structure has a methyl group which can be used. This can be done by identifying the two groups and substituting:

```
>>> abfmpb = csd.molecule('ABFMPB').components[0]  # The crystal structure is a dimer
>>> c1, c10 = abfmpb.atom('C1'), abfmpb.atom('C10')
>>> c3, c9 = abfmpb.atom('C3'), abfmpb.atom('C9')
>>> abfmpb.change_group(c1, c10, abfmpb, c3, c9)
(Atom(C10), Atom(H8), Atom(H9), Atom(H10))
```

Note that copies of the affected atoms are taken, which is why it is safe to use the same molecule as the source of a group.

There is also **`ccdc.molecule.Molecule.add_group()`**, which will copy a set of downstream atoms without changing geometry.

There is a function **`ccdc.molecule.Molecule.fuse_rings()`** which can be used to join two ring systems. Here we can replace the benzene of abfmpb by a naphthalene:

```
>>> # Firstly remove a couple of spare hydrogens
>>> abfmpb = csd.molecule('ABFMPB').components[0]
>>> abfmpb.remove_atoms(a for a in abfmpb.atoms if a.label in ['H12', 'H13'])
>>> # Then remove everything except a benzene from a copy
>>> copy_abfmpb = abfmpb.copy()
>>> c10 = copy_abfmpb.atom('C10')
>>> c1 = copy_abfmpb.atom('C1')
>>> copy_abfmpb.remove_group(c10, c1)
>>> # Remove a spare hydrogen from the copy
>>> copy_abfmpb.remove_atom(copy_abfmpb.atom('H15'))
```

```
>>> # Finally fuse the rings
>>> c13 = abfmpb.atom('C13')
>>> c12 = abfmpb.atom('C12')
>>> c15 = copy_abfmpb.atom('C15')
>>> abfmpb.fuse_rings(c13, c12, copy_abfmpb, c10, c15)
(Atom(C11), Atom(C12), Atom(C13), Atom(C14), Atom(H11), Atom(H14))
```

This will provide a planar ring fusion.

Changing molecular geometry

The API provides methods to change bond lengths, valence and torsion angles and to move the whole molecule.

For example, let us take CUPDAT from the CSD, modify a bond type and make associated edits, such as removing a hydrogen and setting geometries:

```
>>> cupdat = csd.molecule('CUPDAT')
>>> b = cupdat.bond('O1', 'C3')
>>> b.bond_type = 'Double'
>>> cupdat.remove_atom(cupdat.atom('H5'))
>>> cupdat.atom('O1').formal_charge = -1
>>> cupdat.set_bond_length(cupdat.atom('O1'), cupdat.atom('C3'), b.ideal_bond_length)
>>> cupdat.set_valence_angle(cupdat.atom('C1'), cupdat.atom('O1'), cupdat.atom('C3'), 120.0)
>>> cupdat.set_torsion_angle(
...     cupdat.atom('O1'), cupdat.atom('C3'), cupdat.atom('H4'), cupdat.atom('H6'), 180.0
... )
```

Note that the torsion angle set here is an improper torsion, which is handled seamlessly by the API. The **`ccdc.molecule.Bond.ideal_bond_length`** provides a length based on the bond type and the elements it connects.

The whole molecule may be moved around either by translation, rotation about its centre of geometry, by matrices and by Quaternions.

For example to centre a molecule on the origin:

```
>>> aabhtz = csd.molecule('AABHTZ')
>>> cog = aabhtz.centre_of_geometry()
>>> aabhtz.translate([-cog.x, -cog.y, -cog.z])
```

To rotate a molecule about its centre of geometry:

```
>>> aabhtz.rotate([0, 1, 0], 180.)
```

An arbitrary matrix may be applied to the molecule. It is up to the user to avoid inappropriate affine transformations, such as scaling and shearing.

Here we use a numpy array to perform a translation. Numpy may be used to perform matrix multiplications allowing the construction of arbitrary rotations and translations.

```
>>> import numpy
>>> M = numpy.eye(4)
>>> M[0][3] = 1.0
>>> M[1][3] = 2.0
>>> M[2][3] = 3.0
>>> aabhtz.transform(M)
```

A Quaternion may be used instead, which provides a convenient shorthand for a rotation about a particular axis:

```
>>> Q = [1, 0, 1, 0]
>>> aabhtz.apply_quaternion(Q)
```

Coordinates may be transferred from one molecule to another, using an iterable of matched pairs of atoms:

```
>>> aacfaz = csd.molecule('AACFAZ')
>>> aacfaz10 = csd.molecule('AACFAZ10')
>>> aacfaz.set_coordinates(aacfaz10, atoms=zip(aacfaz.atoms[:10], aacfaz10.atoms[:10]))
```

Tidying up

After performing a set of edits it is possible that atom labels will have become duplicated. There is a method, **`ccdc.molecule.Molecule.normalise_labels()`** which will ensure uniqueness of labels.

```
>>> abfmpb.normalise_labels()
```

It may be necessary to reset hydrogens, and to standardise the molecule's bonds after a series of edits:

```
>>> abfmpb.assign_bond_types()
>>> abfmpb.add_hydrogens()
```

While most of the editing operations produce plausible geometries, it may be necessary to minimise the molecule after editing:

```
>>> from ccdc.conformer import MoleculeMinimiser
>>> minimiser = MoleculeMinimiser()
>>> minimised_mol = minimiser.minimise(abfmpb)
```

If a correct protonation state and correct bond-typing has been constructed for a molecule, formal charges for the atoms of the molecule can be assigned using:

```
>>> abfmpb.set_formal_charges()
```

The **`ccdc.crystal.Crystal.molecule`** is writable, so changes to the molecule of a crystal may now be written back to the crystal, and the changes will persist. Obviously this must be used with care, as the edited molecule may not form the same crystal as the original, but this is of considerable use in modelling crystal features. For example to investigate what other solvents might be compatible with a crystal structure one might remove existing solvent molecules and analyse the resulting void volume:

```
>>> abebuf = csd.crystal('ABEBUF')
>>> void_with_solvent = abebuf.void_volume()
>>> abebuf.molecule = abebuf.molecule.heaviest_component
>>> void_without_solvent = abebuf.void_volume()
>>> print 'with %.2f, without %.2f' % (void_with_solvent, void_without_solvent)
with 0.00, without 27.83
```

Search philosophy

The CSD Python API supports four types of search:

- *Substructure searching*
- *Similarity searching*
- *Text numeric searching*
- *Reduced cell searching*

The basic philosophy used to set up and run searches is to:

1. Create a search object
2. Use the **`ccdc.search.Search.search()`** function of the search object to search a specific database

For substructure and similarity searches the database to be searched can be:

- the CSD
- a (multi) molecule file path

- a **ccdc.io** reader (this can be one or more databases; see the note below)
- a Python list of identifiers
- an individual molecule
- an individual crystal

Note

The ability to search a **ccdc.io** reader gives us the capability to search multiple databases simultaneously as we can create readers that contain more than one ASER database.

For example to search the CSD and all the updates one could use the reader specified below.

```
>>> import os.path, glob
>>> from ccdc import io
>>> csd_dir = io.csd_directory()
>>> csd_and_updates = glob.glob(os.path.join(csd_dir, '*.ind'))
>>> csd_and_updates_reader = io.EntryReader(csd_and_updates)
```

Seealso

Working with multiple databases

Because text numeric searches are carried out on fields specific to the CSD these searches can only be performed on the CSD.

The **ccdc.search.Search.search()** function will return a list of **ccdc.search.Search.SearchHit** instances. In some cases these have been specialised for the specific type of search performed:

- **ccdc.search.SubstructureSearch.SubstructureHit**
- **ccdc.search.SimilaritySearch.SimilarityHit**
- **ccdc.search.TextNumericSearch.TextNumericHit**

All hit classes contain an `identifier` as well as attributes to access **ccdc.entry.Entry**, **ccdc.crystal.Crystal** and **ccdc.molecule.Molecule**.

The **ccdc.search.SimilaritySearch.SimilarityHit** additionally contains a `similarity` attribute.

The **ccdc.search.SubstructureSearch.SubstructureHit** has two additional functions:

- **ccdc.search.SubstructureSearch.SubstructureHit.match_atoms()**
- **ccdc.search.SubstructureSearch.SubstructureHit.match_components()**

Furthermore the **ccdc.search.SubstructureSearch.SubstructureHitList** has an additional function for superimposing the hits on the first **ccdc.search.SubstructureSearch.SubstructureHit** in the **ccdc.search.SubstructureSearch.SubstructureHitList**:

- **ccdc.search.SubstructureSearch.SubstructureHitList.superimpose()**

A **ccdc.search.SubstructureSearch.SubstructureHit** may also contain the attributes `measurements` and `constraints` if any geometric measurements/constraints have been added to the **ccdc.search.SubstructureSearch**. These are dictionaries keyed by the name of the measurement or constraint defined on the **ccdc.search.SubstructureSearch**. For more information on measurements and constraints see *Substructure searching with geometric measurements* and *Substructure searching with geometric constraints*.

Seealso

API documentation of the search module

Substructure searching

Introduction

In order to be able to set up a substructure search we will need to import the **ccdc.search** module. Let us also import the **ccdc.io** module to allow us to read in and write out molecules.

```
>>> import ccdc.search
>>> import ccdc.io
```

As a preamble let us set up a variable for a temporary directory.

```
>>> from ccdc.utilities import _test_output_dir
>>> tempdir = _test_output_dir()
```

Let us also get a testosterone molecule out of the CSD.

```
>>> entry_reader = ccdc.io.EntryReader('CSD')
>>> teston10 = entry_reader.molecule('TESTON10')
>>> testosterone = teston10.components[0]
```

Seealso

API documentation of the search module

Setting up a substructure

There are several ways to set up a **ccdc.search.QuerySubstructure**.

It can be created from a molecule.

```
>>> testosterone_substructure = ccdc.search.MoleculeSubstructure(testosterone)
```

It can also be created from a SMARTS string.

```
>>> hydroxyl_substructure = ccdc.search.SMARTSSubstructure("OH")
>>> ketone_substructure = ccdc.search.SMARTSSubstructure("[CD4][CD3](=[OD1])[CD4]")
```

A substructure can also be read in from a ConQuest Connser file.

```
>>> filepath = 'monochloropyridine.con'
```

To achieve this we make use of the **ccdc.search.ConnserSubstructure** class.

```
>>> connser_substructure = ccdc.search.ConnserSubstructure(filepath)
```

Finally, one can create a substructure from scratch.

```
>>> cooh_substructure = ccdc.search.QuerySubstructure()
>>> c = cooh_substructure.add_atom('C')
>>> o1 = cooh_substructure.add_atom('O')
>>> o2 = cooh_substructure.add_atom('O')
>>> b1 = cooh_substructure.add_bond('Double', c, o1)
```

```
>>> b2 = cooh_substructure.add_bond('Single', c, o2)
>>> o1.num_hydrogens = 0
>>> o2.num_hydrogens = 1
```

Setting up and running a substructure search

The substructure instances created earlier can be used to set up substructure searches. Substructure searches can be used to search many different objects. By default a substructure search will search the CSD.

```
>>> substructure_search = ccdc.search.SubstructureSearch()
>>> sub_id = substructure_search.add_substructure(connser_substructure)
>>> hits = substructure_search.search()
>>> print len(hits)
916
```

Note that there may be multiple hits in one structure. The multiplicity of hits can be controlled by optional arguments to the search method: `max_hit_structures` controlling how many structures are returned, and `max_hits_per_structure`. These control how many matches in a structure should be returned. By default these parameters are set to be unlimited (`None`).

```
>>> testosterone_search = ccdc.search.SubstructureSearch()
>>> sub_id = testosterone_search.add_substructure(testosterone_substructure)
>>> hits = testosterone_search.search()
>>> len(hits)
10
>>> len(testosterone_search.search(max_hit_structures=4))
5
>>> len(testosterone_search.search(max_hit_structures=4, max_hits_per_structure=1))
4
>>> unique_hits = testosterone_search.search(max_hits_per_structure=1)
>>> len(unique_hits)
8
```

It is also possible to search the molecules in a (multi) molecule file.

```
>>> file_path = 'testosterone_hits.mol2'
```

This can be achieved using a **`ccdc.io.MoleculeReader`** instance.

```
>>> hydroxyl_search = ccdc.search.SubstructureSearch()
>>> sub_id = hydroxyl_search.add_substructure(hydroxyl_substructure)
>>> hydroxyl_hits = hydroxyl_search.search(ccdc.io.MoleculeReader(file_path))
>>> len(hydroxyl_hits)
20
```

Or, for convenience, by supplying the file path directly to the `search` function.

```
>>> hydroxyl_hits = hydroxyl_search.search(file_path)
>>> len(hydroxyl_hits)
26
```

It is also possible to search an individual molecule.

```
>>> print len( hydroxyl_search.search(testosterone) )
1
```

Or a list of identifiers from the CSD:

```
>>> print len( hydroxyl_search.search(['ABEBUF', 'AABHTZ', 'WOSKOE']) )
3
```

Let us print out the hit identifiers of the original 10 hits identified by the `testosterone_search` search on the CSD.

```
>>> for hit in hits:
...     print hit.identifier
EPITES
ISTEST
ISTEST
TESBRP
TESTHG
TESTOM
TESTOM01
TESTON10
TESTON10
WOSKOE
```

These can be superimposed using the matched atoms of the hits.

```
>>> mols = hits.superimpose()
>>> print len(mols)
10
```

Warning

Notice that we can only superimpose the structures where all atoms have coordinates.

Let us write out the molecules from these hits as a multi-mol2 file.

```
>>> output_file = os.path.join(tempdir, 'testosterone_hits.mol2')
>>> with ccdc.io.MoleculeWriter(output_file) as writer:
...     for m in mols:
...         writer.write(m)
```

We can also write the result data into a ConQuest to Mercury interchange file, so that the results, with constraints and measurements may be analysed in the data analysis module of Mercury.

```
>>> hits.write_c2m_file(os.path.join(tempdir, 'testosterone_hits.c2m'))
```

We can find the matched atoms for each hit, where the hit atoms have coordinates:

```
>>> print hits[0].match_atoms()  # doctest: +ELLIPSIS
[Atom(C1), Atom(C2), Atom(C3), ...]
```

All forms of search hit support molecule, crystal and entry properties:

```
>>> entry = hits[0].entry
>>> print entry.chemical_name == u'17\u03b1-Hydroxyandrost-4-en-3-one'
True

>>> crystal = hits[0].crystal
>>> print crystal.spacegroup_symbol
P212121

>>> molecule = hits[0].molecule
```

```
>>> print len(molecule.atoms)
49
```

> ### Note
>
> The chemical name from the entry **`ccdc.entry.Entry.chemical_name`** is in Unicode format and *\u03b1* is the encoding for the 'GREEK SMALL LETTER ALPHA' www.fileformat.info/info/unicode/char/3b1/index.htm. For more information on how to work with Unicode in Python please see docs.python.org/2.7/howto/unicode.html

Substructure searching with geometric measurements

It is possible to add geometric measurements to **`ccdc.search.SubstructureSearch`** searches.

The geometric measurements are added to the **`ccdc.search.SubstructureSearch`** using the functions:

- **`ccdc.search.SubstructureSearch.add_distance_measurement()`**
- **`ccdc.search.SubstructureSearch.add_angle_measurement()`**
- **`ccdc.search.SubstructureSearch.add_torsion_angle_measurement()`**

Say, for example, that we were interested in understanding the intra-molecular geometry of an aromatic methoxy group. In particular how the preference of the methoxy group to lie in the plane of the aromatic ring affects the Ph-C-O angle.

Let us first create the substructure of interest using a **`ccdc.search.SMARTSSubstructure`**. The substructure can then be used to to set up a **`ccdc.search.SubstructureSearch`**.

```
>>> ar_methoxy_sub = ccdc.search.SMARTSSubstructure('[CH3:1][O:2][c:3]1[cH:4]ccc[cH:5]1')
>>> ar_methoxy_search = ccdc.search.SubstructureSearch()
>>> ar_methoxy_sub_id = ar_methoxy_search.add_substructure(ar_methoxy_sub)
```

We can now add the measurements of interest using the indices of the atoms of interest in the SMARTS pattern.

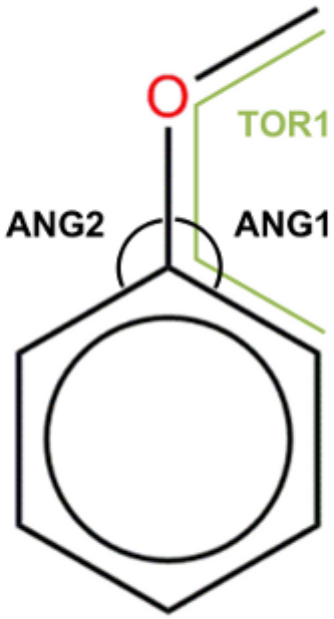


Figure illustrating the aromatic methoxy query.

```
>>> ar_methoxy_search.add_angle_measurement('ANG1',
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(2),
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(3),
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(4))
>>> ar_methoxy_search.add_angle_measurement('ANG2',
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(2),
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(3),
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(5))
>>> ar_methoxy_search.add_torsion_angle_measurement('TOR1',
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(1),
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(2),
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(3),
...     ar_methoxy_sub_id, ar_methoxy_sub.label_to_atom_index(4))
```

Note

Here we are making use of the **`ccdc.search.SMARTSSubstructure.label_to_atom_index()`** function to convert the reaction SMARTS labels into zero based substructure atom indices.

```
>>> ar_methoxy_sub.label_to_atom_index(1)
0
```

Now we are ready to carry out the search. Because this aromatic methoxy substructure is quite common in the CSD, we will limit our maximum number of hits to 200. Further, to avoid bias by picking multiple observations from the same structure we will limit the number of hits per structure to 1.

```
>>> ar_methoxy_hits = ar_methoxy_search.search(max_hit_structures=200, max_hits_per_structure=1)
>>> len(ar_methoxy_hits)
200
```

To get the data out of the list of **`ccdc.search.SubstructureHit`** instances we can make use of list comprehension and Python's built in `zip` functionality.

```
>>> measurements = [ (h.measurements['ANG1'],
...                   h.measurements['ANG2'],
...                   abs(h.measurements['TOR1']))
...                  for h in ar_methoxy_hits ]
>>> ang1, ang2, abstor1 = zip(*measurements)
```

Seealso

For more information on Python's built in `zip` function please see the Python on-line documentation docs.python.org/2/library/functions.html#zip.

Finally, we can plot the data using matplotlib.

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(ang1, ang2, c=abstor1, vmin=0, vmax=180)  # doctest: +ELLIPSIS +SKIP
<matplotlib.collections.PathCollection object at ...>
>>> cbar = plt.colorbar() # doctest: +SKIP
>>> cbar.set_label('TOR1') # doctest: +SKIP
>>> plt.title('Aromatic methoxy geometry')  # doctest: +ELLIPSIS +SKIP
```

```
<matplotlib.text.Text object at ...>
>>> plt.xlabel('ANG1')  # doctest: +ELLIPSIS +SKIP
<matplotlib.text.Text object at ...>
>>> plt.ylabel('ANG2')  # doctest: +ELLIPSIS +SKIP
<matplotlib.text.Text object at ...>
>>> plt.show() # doctest: +SKIP
```

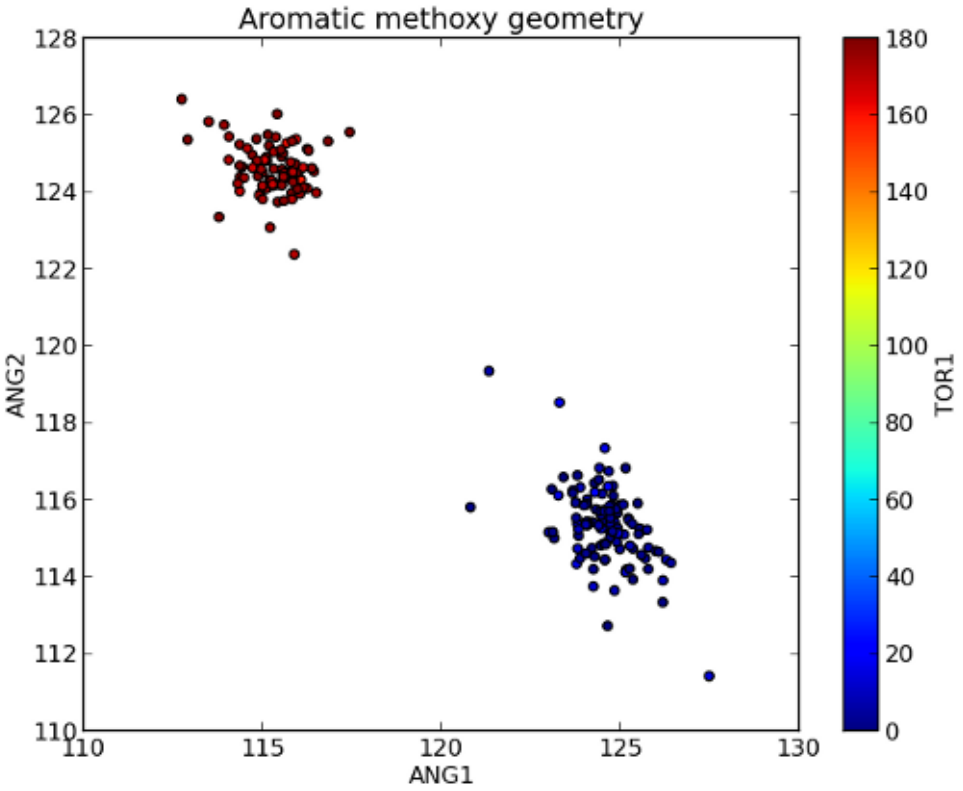


Figure illustrating the effect of steric crowding induced by the methyl group lying in the plane of the aromatic ring for a methoxy group attached to an aromatic ring. Note the deviation from the "ideal" of 120°.

Substructure searching with geometric constraints

It is also possible to add geometric constraints to substructure searches. These can be added using:

- **`ccdc.search.SubstructureSearch.add_distance_constraint()`**
- **`ccdc.search.SubstructureSearch.add_angle_constraint()`**
- **`ccdc.search.SubstructureSearch.add_torsion_angle_constraint()`**

Suppose, for example, that we wanted to understand the interaction geometry of an aromatic iodine and the nitrogen atom of a pyridine ring. Specifically, does the C-I···N angle tend towards zero as the I···N distance becomes shorter?

First of all let us define the substructures to search for.

```
>>> ar_I_sub = ccdc.search.SMARTSSubstructure('Ic1ccccc1')  # I: index 0
>>> pyridine_sub = ccdc.search.SMARTSSubstructure('n1ccccc1')  # n: index 0
```

Using these substructures we can set up the search with a distance constraint between the iodine and the pyridine nitrogen atoms.

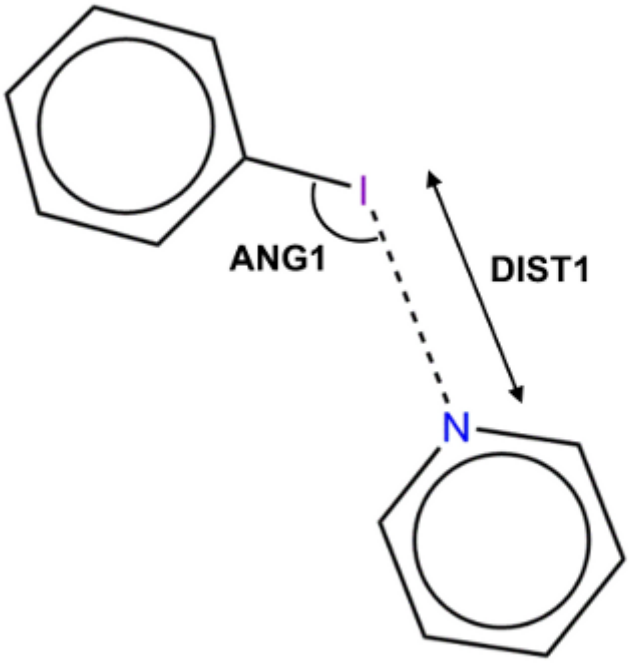


Figure illustrating the aromatic iodine ··· pyridine query.

```
>>> halogen_bond_search = ccdc.search.SubstructureSearch()
>>> ar_I_sub_id = halogen_bond_search.add_substructure(ar_I_sub)
>>> pyridine_sub_id = halogen_bond_search.add_substructure(pyridine_sub)
```

Note that at this point we have added two substructures to the search and we can add a distance constraint between them. This requires us to specify both the substructure and atom identifiers of interest. Incidentally this is why the **`ccdc.search.SubstructureSearch.add_substructure()`** function returns the substructure identifier.

```
>>> halogen_bond_search.add_distance_constraint('DIST1',
...     ar_I_sub_id, 0,
...     pyridine_sub_id, 0,
...     (0.0, 3.4),  # distance constraint range
...     'Intermolecular')
```

Note

We could have specified the distance with respect to the van der Waals radii of the atoms using the by setting the *vdw_corrected* parameter of the **`ccdc.search.SubstructureSearch.add_distance_constraint()`** function to *True*.

Rather than just measure the C-I···N angle we can add it as a angular constraint, ensuring that it be greater than 120° for a match.

```
>>> halogen_bond_search.add_angle_constraint('ANG1',
...     ar_I_sub_id, 1,  # the carbon that the I is attached to
...     ar_I_sub_id, 0,
...     pyridine_sub_id, 0,
...     (120.0, 180.0))  # the angle constraint range
```

We can now carry out the search.

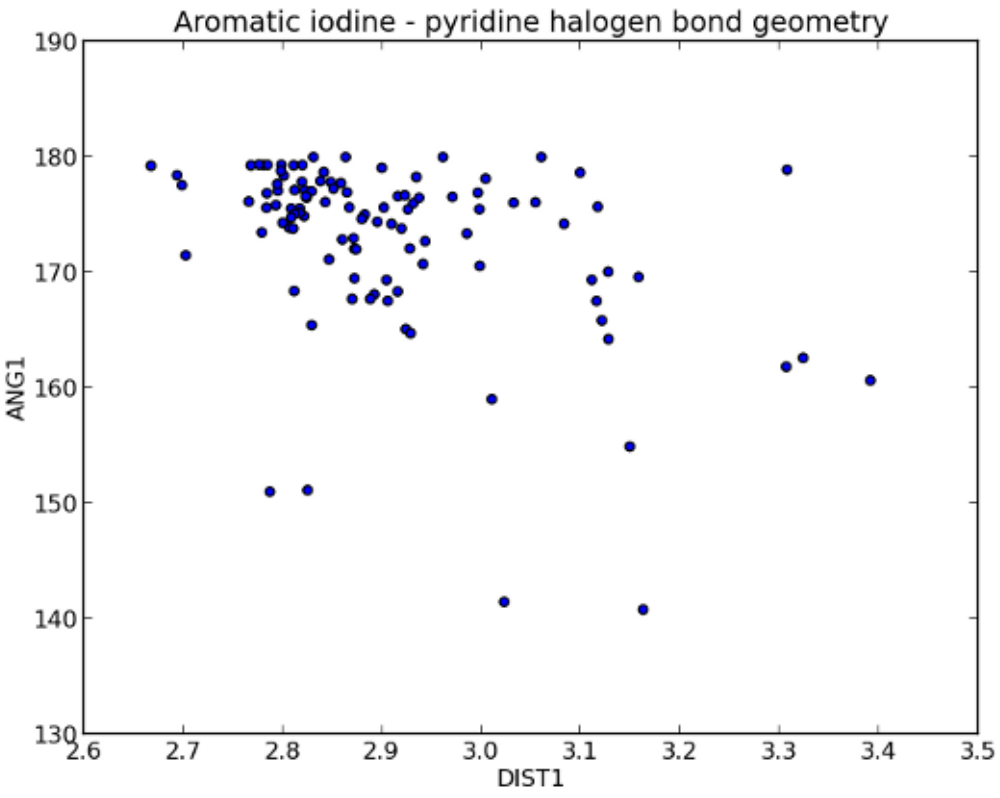
```
>>> halogen_bond_hits = halogen_bond_search.search(max_hits_per_structure=1)
>>> len(halogen_bond_hits)
184
```

To get the data out of the list of **`ccdc.search.SubstructureHit`** instances we can make use of list comprehension and Python's built in `zip` functionality.

```
>>> dist1_ang1 = [(h.constraints['DIST1'], h.constraints['ANG1'])
...               for h in halogen_bond_hits ]
>>> dist1, ang1 = zip(*dist1_ang1)
```

We can use matplotlib to check for any geometric preferences of the aromatic iodine ⋯ pyridine halogen bond.

```
>>> plt.clf()  # Clear the figure from the previous plot # doctest: +SKIP
>>> plt.scatter(dist1, ang1)  # doctest: +ELLIPSIS +SKIP
<matplotlib.collections.PathCollection object at ...>
>>> plt.title('Aromatic iodine - pyridine halogen bond geometry')  # doctest: +ELLIPSIS +SKIP
<matplotlib.text.Text object at ...>
>>> plt.xlabel('DIST1')  # doctest: +ELLIPSIS +SKIP
<matplotlib.text.Text object at ...>
>>> plt.ylabel('ANG1')  # doctest: +ELLIPSIS +SKIP
<matplotlib.text.Text object at ...>
>>> plt.show() # doctest: +SKIP
```



Plotting the angle versus the distance reveals that there a weak negative correlation, i.e., as the contact distance becomes shorter the angle tends towards 180°.

Search filters

It is possible to constrain the hits of a substructure search by various criteria. This is achieved by setting constraints on the **ccdc.search.Search.Settings**. For example to restrict a search to provide only organic compounds, with no disorder and with a good R-factor,

```
>>> halogen_bond_search.settings.only_organic = True
>>> halogen_bond_search.settings.no_disorder = 'all'
>>> halogen_bond_search.settings.max_r_factor = 5.0
```

Then the search may be performed as before. There are also methods to ensure that particular elements are not present in a hit, or to ensure that particular elements must be present in a hit. For example to prohibit hits containing metalloid elements (which may be present in organic compounds):

```
>>> halogen_bond_search.settings.must_not_have_elements = ['B','Si','Ge','As','Sb','Te','At']
```

In additon to such substructure searches, these search filters can also be applied to **ccdc.search.TextNumericSearch**, **ccdc.search.SimilaritySearch** and **ccdc.search.ReducedCellSearch** searches.

Disorder

To configure how disorder of structures mediates the search, **ccdc.search.Search.Settings.no_disorder** may take any of three values:

```
- None (or anything that evaluates to False) to indicate no filtering of any disordered structures,
- 'all' to indicate filtering of structures with any disordered atoms, or
- 'Non-hydrogen' (or any string apart from 'all') to indicate filtering of structures with heavy atom disorder.
```

The last option, 'Non-hydrogen', is compatible with the ConQuest 'no disorder' selector.

Substructure Screens

Where there is a large multi-molecule 'sdf' or 'mol2' format file which one might wish to search more than once, for example a docking result file or a compound catalogue, then it is possible to calculate a set of screening fingerprints for the molecules of the database, and use them to speed up substructure searches. The calculation of these screening fingerprints is computationally very expensive, so this approach is only beneficial when many substructure searches will be performed on the database. These screens are pre-calculated within Aser format databases and the CSD.

If desired then screening fingerprints may be defined for a reader. If so, and if the reader has an attribute 'substructure_screen', this will be used in a SubstructureSearch to accelerate any searches.

I will exemplify the process on a trivial database. Firstly declare a reader, calculate the screens, and then write them to a file.

```
>>> reader = ccdc.io.EntryReader(locate_file('short.gcd'))
>>> screens = ccdc.search.SubstructureSearch.Screen.create(reader)
>>> screens.write(os.path.join(temp, 'screens.dat'))
```

Now when we wish to search the database we can read and attach the screen data to the database:

```
>>> screens = ccdc.search.SubstructureSearch.Screen.from_file(os.path.join(temp, 'screens.dat'))
>>> reader.substructure_screen = screens
```

This will automatically be used by any subsequent substructure search of the reader. The degree of screening may be assessed for an individual search:

```
>>> sub_search = ccdc.search.SubstructureSearch()
>>> sub_index = sub_search.add_substructure(ccdc.search.SMARTSSubstructure('S(=O)(=O)'))
>>> candidates = screens.candidates(sub_search)
>>> print '%d hits out of %d' % (len(candidates), len(reader))
2 hits out of 12
>>> print ' '.join(reader[i].identifier for i in candidates)
ACAPEA ACIFAT
```

```
>>> hits = sub_search.search(reader)
>>> print ' '.join(h.identifier for h in hits)
ACAPEA ACAPEA ACAPEA ACAPEA ACIFAT
```

The substructure search will be approximately six-fold faster using the screen.

Similarity searching

Introduction

In order to be able to set up searches we will need to import the **`ccdc.search`** module. Let us also import the **`ccdc.io`** module to allow us to read in and write out molecules.

```
>>> import ccdc.search
>>> import ccdc.io
```

As a preamble let us also set up a variable for a temporary directory and a file path to a testosterone molecule.

```
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
>>> filepath = 'testosterone.mol2'
>>> filepath = locate_file(filepath)
```

To get access to the molecule in the testosterone mol2 file we make use of a **`ccdc.io.MoleculeReader`**.

```
>>> reader = ccdc.io.MoleculeReader(filepath)
>>> testosterone = reader[0]
```

Seealso

API documentation of the search module

Similarity search

To run a similarity search one must first create a **`ccdc.search.SimilaritySearch`** whose initialiser takes a **`ccdc.molecule.Molecule`** and a similarity threshold between 0.0 and 1.0. By default the similarity threshold is set to 0.7.

```
>>> similarity_query = ccdc.search.SimilaritySearch(testosterone)
```

The similarity search can then be run by making use of the `search()` function.

```
>>> sim_hits = similarity_query.search()
>>> print len(sim_hits)
504
```

To reduce the number of hits we can increase the similarity threshold.

```
>>> similarity_query.threshold = 0.9
>>> sim_hits = similarity_query.search()
>>> print len(sim_hits)
66
```

An alternative approach to reducing the number of hits is to constrain the number of hits to return.

```
>>> sim_hits = similarity_query.search(max_hit_structures=10)
>>> print len(sim_hits)
10
```

Let us find out what these structures are and what their similarity to the query is.

```
>>> for hit in sim_hits:
...     print '%9s: %.2f' % (hit.identifier, hit.similarity)
   BAWMAN: 1.00
   BEJVAN: 1.00
   BOKVUS: 1.00
   CERVAX: 1.00
   DIGRIV: 1.00
   EPITES: 1.00
   HANSTO: 1.00
   HXANDO: 1.00
   ISTEST: 1.00
   MHBFLU: 1.00
```

Similarity searches allow all forms of search filter given by **ccdc.search.Search.Settings**. See *Search filters* for examples of use.

Similarity queries allow all the forms of search that a **ccdc.search.SubstructureSearch** does.

Seealso

For more information please see *Setting up and running a substructure search*

For example we can use an instance of a **ccdc.molecule.Molecule** directly.

```
>>> file_path = 'ABEBUF.mol2'
```

To get access to the first molecule in the ABEBUF.mol2 file we make use of the **ccdc.io.MoleculeReader**.

```
>>> h = similarity_query.search_molecule(ccdc.io.MoleculeReader(file_path)[0])
>>> print h.identifier, h.similarity
ABEBUF 0.105911330049
```

Text-numeric searching

Introduction

The CSD supports a certain amount of text or numeric searching of entries.

Firstly, let us import the necessary module.

```
>>> from ccdc.search import TextNumericSearch
```

Warning

This class may only be used to search the CSD.

Seealso

API documentation of the search module

Searching the CSD using text numeric searches

Let us start off by creating an empty query.

```
>>> query = TextNumericSearch()
```

Suppose that we wanted to search a particular journal. The first step to this would be to make sure that the string used to specify the journal was valid/represented in the CSD.

```
>>> query.is_journal_valid('J.Med.Chem.')
True
>>> query.is_journal_valid('Journal of Medicinal Chemistry')
False
```

It is possible to programatically search known journals if you have a vague idea of some part of the title:

```
>>> print '\n'.join(k for k in query.journals if 'med.chem' in k.lower())
Bioorg.Med.Chem.Lett.
Eur.J.Med.Chem.
Med.Chem.Res.
Indian J.Chem.,Sect.B:Org.Chem.Incl.Med.Chem.
Zhongguo Yaowu Huaxue Zazhi(Chin.)(Chin.J.Med.Chem.)
J.Enzyme Inhib.Med.Chem.
J.Med.Chem.
Org.Med.Chem.Lett.
Bioorg.Med.Chem.
Curr.Top.Med.Chem.
Med.Chem.
ACS.Med.Chem.Lett.
```

If we have exact details of a citation we can find the structures published in the paper.

```
>>> query = TextNumericSearch()
>>> query.add_citation(journal='Organometallics', year=2000, volume=19, first_page=3354)
>>> print '\n'.join(h.identifier for h in query.search())
XAGMUN
XAGNAU
XAGNEY
XAGNIC
XAGNOI
XAGNUO
XAGPAW
```

Not all the fields of a citation are required. One can, for example, find out how many structures were published by J.Med.Chem. in 2008:

```
>>> query = TextNumericSearch()
>>> query.add_citation(journal='J.Med.Chem.', year=2008)
>>> print len('\n'.join(h.identifier for h in query.search()))
796
```

Or even histogram the growth of the CSD over the years.

```
>>> nhits = []
>>> tot = 0
>>> for i in range(1921, 2015):
...     query = TextNumericSearch()
```

```
...     query.add_citation(year=i)
...     tot += len(query.search())
...     nhits.append((i, tot))
>>> print nhits  # doctest: +ELLIPSIS
[(1921, 0), (1922, 0), (1923, 2), ... (2012, 665255), (2013, 712112), (2014, 760326)]
>>> print tot
760326
```

There are other available text numeric searches. The code snippet below illustrates a search by chemical name.

```
>>> query = TextNumericSearch()
>>> query.add_synonym('aspirin')
>>> print '\n'.join(h.identifier for h in query.search())
ACMEBZ
ACSALA
ACSALA01
ACSALA02
ACSALA03
ACSALA04
ACSALA05
ACSALA06
ACSALA07
ACSALA08
ACSALA09
ACSALA10
ACSALA11
ACSALA12
ACSALA13
ACSALA14
ACSALA15
ACSALA16
ACSALA17
ACSALA18
ACSALA19
ACSALA20
ACSALA21
ARIFOX
BEHWOA
EYOMEL
EYOMIP
EYOMOV
EYOMUB
EYONAI
HUNJEH
HUPPOX
IBOBUY
IBOBUY01
IBOCEJ
IBOCEJ01
IBOCOT
IBOCOT01
JIRNEE
KEWNOQ
LAJVUO01
NINFUN
NUKXOH
NUWTIJ01
NUWTOP01
NUWTOP02
TORQUM02
```

```
ACSALA22
HUPPOX01
KEWNOQ01
```

It is also possible to find out how many structures Greg Shields published.

```
>>> query.clear()
>>> query.add_author('G.P.Shields')
>>> print len(query.search())
114
```

Text queries may take an optional mode, which will modify the search. Available modes are accessible from the query:

```
>>> print '\n'.join(k for k in query.modes)
not_null
anywhere
separate
is_null
exact
start
start_of_word
```

'Separate' means a separate, space delimited word within the field.

```
>>> query.clear()
>>> query.add_color('red', mode='exact')
>>> print 'There are %d red compounds in the CSD' % len(query.search())
There are 71098 red compounds in the CSD
```

Text-numeric searches allow all forms of search filter given by **ccdc.search.Search.Settings**. See *Search filters* for examples of use. To search, for example, for organic red compounds we would specify:

```
.. code-block:: python
```

```
>>> query.settings.only_organic = True
>>> print 'There are %d organic red compounds in the CSD' % len(query.search())
There are 10497 organic red compounds in the CSD
```

```
>>> query.settings.only_organic = False
```

How many entries do not have a doi?

```
>>> query.clear()
>>> query.add_doi('', mode='is_null')
>>> print len(query.search())
119664
```

How many entries contain the word "dihydrofolate"?

```
>>> query.clear()
>>> query.add_all_text('dihydrofolate', mode='exact')
>>> print len(query.search())
36
```

There is also an optional argument that can be used to ignore non alpha numeric parts of a hit. Let us illustrate this with a compound name search.

```
>>> query.clear()
>>> query.add_compound_name('azabicyclononane')
>>> hits = query.search()
>>> len(hits)
1
>>> print hits[0].entry.chemical_name
Dimethyl (6RS,8RS)-8-phenyl-9-oxa-1-azabicyclononane-5,5-dicarboxylate
>>> query.clear()
>>> query.add_compound_name('azabicyclononane', ignore_non_alpha_num=True)
>>> hits = query.search()
>>> len(hits)
535
>>> print hits[-1].entry.chemical_name
4-methylene-2-((4-methylphenyl)sulfonyl)octahydro-1H-cyclopenta[c]pyridine
```

Queries may be joined with an implicit boolean 'and'.

```
>>> query.clear()
>>> query.add_author('F.H.Allen')
>>> query.add_author('J.Trotter')
>>> print len(query.search())
9
```

A numeric query may take a pair of values, interpreted as an inclusive range. This can be used to, for example, find out if there are any recent aspirin structures.

```
>>> query.clear()
>>> query.add_compound_name('aspirin')
>>> query.add_citation(year=[2011,2013])
>>> print '\n'.join(h.identifier for h in query.search())
ACSALA19
ACSALA20
ACSALA21
ARIFOX
EYOMEL
EYOMIP
EYOMOV
EYOMUB
EYONAI
IBOBUY
IBOBUY01
IBOCEJ
IBOCEJ01
IBOCOT
IBOCOT01
KICVUP
NINFUN
NUWTIJ01
NUWTOP01
NUWTOP02
```

The difference between 'exact' and 'anywhere' is that an 'exact' query of "cat" would only match "cat", but an 'anywhere' query would also match "catty". Let us illustrate this with a search on compound name. The default behaviour is to use 'anywhere' mode.

```
>>> query.clear()
>>> query.add_compound_name('acetylcholine')
>>> print len(query.search())
20
```

When using the 'exact' mode we get two fewer hits.

```
>>> query.clear()
>>> query.add_compound_name('acetylcholine', mode='exact')
>>> print len(query.search())
18
```

Suppose that we were interested in finding polymorphic ibuprofens. Below is a code snippet illustrating one way of performing such a search.

```
>>> query.clear()
>>> query.add_synonym('ibuprofen', 'exact')
>>> print len(query.search())
43
>>> query.add_polymorph('', 'not_null')
>>> print '\n'.join(h.identifier for h in query.search())
IBPRAC
IBPRAC01
IBPRAC02
IBPRAC03
IBPRAC04
IBPRAC05
IBPRAC06
IBPRAC07
IBPRAC08
IBPRAC09
IBPRAC10
IBPRAC11
IBPRAC12
IBPRAC13
IBPRAC14
IBPRAC15
IBPRAC16
IBPRAC17
IBPRAC18
IBPRAC19
```

It is also possible to perform a search using the bioactivity field in the CSD.

```
>>> query.clear()
>>> query.add_bioactivity('antiinflammatory')
>>> print len(query.search())
482
>>> query.add_bioactivity('analgesic')
>>> print len(query.search())
131
```

Let us look for CSD entries that have the exact words "backbone" and "ligand" in their disorder description.

```
>>> query.clear()
>>> query.add_disorder('backbone', 'exact')
>>> query.add_disorder('ligand', 'exact')
>>> hits = query.search()
>>> print '\n'.join(h.identifier for h in hits)
AJEGIF
COSZEP
DASMOB
HINPAV
YIWRIF
YIWRIG
```

To show what this means let us print the disorder details of the entry AJEGIF. We can obtain the entry directly from the hit.

```
>>> ajegif = hits[0].entry
>>> print ajegif.disorder_details  # doctest: +NORMALIZE_WHITESPACE
The ligand backbone exhibits a racemic twinning disorder in which
the molecule is disordered over two sites in a 3:1 ratio. One isopropyl
C atom is disordered over two sites with occupancies 0.51:0.49.
```

For convenience hits also have properties for the crystal and the molecule of a hit.

A **ccdc.search.TextNumericSearch** will display its component queries in a human readable form:

```
>>> query.clear()
>>> query.add_compound_name('aspirin')
>>> query.add_citation(year=[2011,2013])
>>> print('\n'.join(q for q in query.queries))
Compound name aspirin anywhere
Journal year in range 2011-2013
```

Reduced cell searching

Introduction

Reduced cell searching of crystal structures is of practical use when collecting crystallographic data. Before collecting a full data set one can use a reduced cell search using the pre-experiment unit cell against the CSD for the purpose of:

- Matching a known crystal structure against the cell dimensions previously published.
- Checking that a new crystal sample you are looking to collect has not been published before.
- Helping to check that starting materials or a reaction by-product have not been crystallised by accident.

The reduced cell searching functionality is located in the **ccdc.search** module.

```
>>> from ccdc.search import ReducedCellSearch
```

Let us also get access to the AACMHX10 crystal.

```
>>> from ccdc.io import EntryReader
>>> entry_reader = EntryReader('CSD')
>>> aacmhx10 = entry_reader.crystal('AACMHX10')
```

Use of the Niggli reduced cell (commonly termed "the reduced cell") is problematic due to numerical instabilities. While the reduced cell can be uniquely specified for any given crystal lattice, the reuced cell angles can change greatly with only small changes in the lattice parameters. A reduced unit cell search algorithm does not suffer from this problem. The reduced cell search algorithm employed by the CSD Python API is the same as that used in WebCSD which relies on "nearly Buerger reduced cells". In this method the reduced cells and a set of closely related cells are determined and this extended set of cell parameters is used to evaluate whether any pair of crystals share the same, or essentially the same, reduced cell.

Performing reduced cell searches

Here we will illustrate the reduced cell searching functionality using the cell parameters from AACMHX10.

```
>>> print aacmhx10.cell_lengths
CellLengths(a=24.157, b=16.758, c=8.535)
>>> print aacmhx10.cell_angles
CellAngles(alpha=90.0, beta=90.0, gamma=90.0)
```

```
>>> print aacmhx10.lattice_centring
primitive
```

To perform a reduced cell search let us create an instance of **ccdc.search.ReducedCellSearch**.

```
>>> query = ReducedCellSearch.Query(aacmhx10.cell_lengths,
...                                 aacmhx10.cell_angles,
...                                 aacmhx10.lattice_centring)
>>> searcher = ReducedCellSearch(query)
>>> hits = searcher.search()
```

Let us analyse the hits.

```
>>> len(hits)
25
>>> h = hits[0]
>>> print h.identifier, h.crystal.cell_lengths, h.crystal.cell_angles, h.crystal.lattice_centring # doctest: +NORMALIZE_WHITESPACE
    AACMHX10
    CellLengths(a=24.157, b=16.758, c=8.535)
    CellAngles(alpha=90.0, beta=90.0, gamma=90.0)
    primitive
```

It is also possible to set up reduced cell searches from a **ccdc.crystal.Crystal** instance using the **ccdc.search.ReducedCellSearch.CrystalQuery** class, as well as from CellCheckCSD XML files using either the **ccdc.search.ReducedCellSearch.XMLQuery** class or the **ccdc.search.ReducedCellSearch.from_xml()** function.

Seealso

More information about CellCheckCSD can be found on the CCDC website www.ccdc.cam.ac.uk/Solutions/FreeSoftware/Pages/CellCheckCSD.aspx

Note

The settings used in the reduced cell search can be modified in the **ccdc.search.ReducedCellSearch.Settings**.

Reduced cell searches allow all forms of search filter given by **ccdc.search.Search.Settings**. See *Search filters* for examples of use.

Seealso

API documentation of the search module

Conformer generation and molecular minimisation

Note

The **ccdc.conformer** module is available only to CSD-Discovery, CSD-Materials and CSD-Enterprise users.

Introduction

The **ccdc.conformer** module can be used to generate a diverse ensemble of conformers that are optimised to reflect the geometrical preferences of molecules observed in the CSD.

The methodology requires an input molecule where all atoms have 3D coordinates. The following steps are then performed:

- The bond distances and angles are optimised according to data from the CSD
- Conformational space is sampled using rotamer distributions and ring templates derived from the CSD and the generated conformers are scored
- A diverse subset is selected

The **ccdc.conformer.ConformerGenerator** is available in the **ccdc.conformer** module. Let us import this module.

```
>>> from ccdc import conformer
```

Let us also import the **ccdc.io** module so that we can read in a molecule of interest and define the path to a molecule file of interest, in this instance a CDK2 inhibitor.

```
>>> from ccdc import io
>>> filepath = '2vtp-lig.mol2'
```

In this instance we will get the first molecule in the file.

```
>>> mol_reader = io.MoleculeReader(filepath)
>>> mol = mol_reader[0]
>>> mol_reader.close()
```

Seealso

API documentation of the conformer module

Conformer generation

To generate conformers we create a **ccdc.conformer.ConformerGenerator**.

```
>>> conformer_generator = conformer.ConformerGenerator()  # doctest: +SKIP
```

Note

There are optional parameters for finer control of the **ccdc.conformer.ConformerGenerator**; please see the API documentation for further detail.

The **ccdc.conformer.ConformerGenerator** has a number of settings that can be customised, for example the maximum number of conformers to generate. By default the conformer generator will try to generate 200 conformers. Let us change this number to 25 instead.

```
>>> conformer_generator.settings.max_conformers  # doctest: +SKIP
200
>>> conformer_generator.settings.max_conformers = 25  # doctest: +SKIP
```

Note

The **ccdc.conformer.ConformerSettings.max_conformers** setting specifies the maximum number of conformers to generate. If one specified a large number and conformational space is exhausted before this number is reached a smaller number of conformers will be generated.

To generate a set of conformers for a molecule one simply calls the **ccdc.conformer.ConformerGenerator.generate()** function.

```
>>> conformers = conformer_generator.generate(mol)  # doctest: +SKIP
>>> len(conformers)  # doctest: +SKIP
25
```

> ***Note***
>
> The **ccdc.conformer.ConformerGenerator.generate()** function can also take a list of molecules which may give some performance improvement when working on large numbers of molecules.

The **ccdc.conformer.ConformerGenerator.generate()** function returns a **ccdc.conformer.ConformerHitList** object, which behaves like a Python list. The **ccdc.conformer.ConformerHitList** contains attributes relating to overall performance of the run. For example whether or not the sampling limit was reached and how many rotamers had no observations in the CSD.

```
>>> conformers.sampling_limit_reached  # doctest: +SKIP
False
>>> conformers.n_rotamers_with_no_observations  # doctest: +SKIP
0
```

An individual **ccdc.conformer.ConformerHit** contains a **ccdc.molecule.Molecule** as well the **ccdc.conformer.ConformerHit.normalised_score** - a value between 0.0 (most probable) and 1.0 (least probable).

```
>>> most_probable_conformer = conformers[0]  # doctest: +SKIP
>>> most_probable_conformer.molecule    # doctest: +SKIP +ELLIPSIS
<ccdc.molecule.Molecule ...>
>>> '%.2f' % most_probable_conformer.normalised_score  # doctest: +SKIP
'0.00'
```

Molecular minimisation

The **ccdc.conformer** module also contains functionality for optimising a molecule to the geometrical preferences of bond distances and valence angles observed in the CSD.

> ***Note***
>
> By default this is the first step in the conformer generator.

The geometrical optimisation makes use of the Tripos force field functional forms and, where available, equilibrium bond distances and valence angles are parameterised using data obtained from the CSD.

```
>>> molecule_minimiser = conformer.MoleculeMinimiser()  # doctest: +SKIP
>>> minimised_mol = molecule_minimiser.minimise(mol)  # doctest: +SKIP
```

To compare the two molecules we can make use of the **ccdc.descriptors.MolecularDescriptors.rmsd()** function.

```
>>> from ccdc.descriptors import MolecularDescriptors
>>> round( MolecularDescriptors.rmsd(mol, minimised_mol), 3)  #  doctest: +SKIP
0.662
```

Field-based virtual screening

> **Note**
>
> The **ccdc.screening** module is under development – currently available only to associated collaborators.

Introduction

The **ccdc.screening** module can be used to screen a library of compounds against a pharmacophore query obtained from one or multiple overlaid ligands. The algorithm generalises the 3D pharmacophore definition using atom property fields that are created around the query based on user-defined atom types and potentials.

Three main steps are performed:

- Generation of a field potential from the query and creation of fitting points in hot-spots
- Global optimisation of the translation and rotation of each ligand by generating conformer libraries and then fitting the ligand atoms to precalculated fitting points
- Scoring of each screened ligand with numerical gradients.

The methodology requires two input files (*i.e.*, a query and the molecules to screen) where all atoms have 3D coordinates.

Screening Workflow

The **ccdc.screening.Screener** is available in the **ccdc.screening** module. Let us import this module.

```
>>> from ccdc.screening import Screener
```

Let us also import the **ccdc.io** module so that we can read in the input files.

```
>>> from ccdc import io

>>> query_file = 'P28845.sdf'
>>> screen_set_file = 'P28845_actives.sdf'

>>> query_filepath = locate_file(query_file)
>>> screen_set_filepath = locate_file(screen_set_file)

>>> query = [m for m in io.MoleculeReader(query_filepath)]
>>> screen_set = [m for m in io.MoleculeReader(screen_set_filepath)]
```

The **ccdc.screening.Screener.Settings** allows the output directory to be defined. This is where all generated files will be stored.

```
>>> import os

>>> settings = Screener.Settings()  # doctest: +SKIP
>>> settings.output_directory = os.path.join(os.getcwd(), "Screen_data")  # doctest: +SKIP
```

It is also possible to define the parameter directory if one wishes to use customised atom types definition and potentials. Here we use the default.

The **ccdc.screening.Screener.screen()** performs the field-based ligand screening, returning a **ccdc.screening.Screener.ScreenHitList**. Let us generate the field potentials around the query.

```
>>> screener = Screener(query, settings=settings)  # doctest: +SKIP
```

Molecules in screen_set are then screened using their input conformation.

```
>>> results = screener.screen([ [m] for m in screen_set])  # doctest: +SKIP
```

It is also possible to use conformer libraries and select the conformation of each ligand that best matches the query fields.

Seealso

For more information on the conformer generation please see the *descriptive conformer generation and molecular minimisation documentation* and the **ccdc.conformer.ConformerGenerator** API documentation.

Each result is represented in a **ccdc.screening.Screener.ScreenHitList.ScreenHit** instance.

```
>>> scores = sorted([(r.score, r.identifier) for r in results])  # doctest: +SKIP
>>> from ccdc.io import MoleculeWriter
>>> molwriter = MoleculeWriter('P28845_results.mol2')  # doctest: +SKIP
>>> for r in results:  # doctest: +SKIP
...     molwriter.write(r.molecule)
```

Docking

Note

The **ccdc.docking** module is under development – currently available only to associated collaborators.

Introduction

The **ccdc.docking** module provides an API to docking.

The module contains a single class, **ccdc.docking.Docker** which, like other classes of the CSD Python API contains a nested **ccdc.docking.Docker.Settings** which will be used to specify the desired docking. Once the nested settings class is appropriately configured, **ccdc.docking.Docker.dock()** can be called to perform the docking.

Setting up the docking

Let us import the appropriate module:

```
>>> from ccdc.docking import Docker
```

A docking requires one or more protein files, one or more ligand files, and a binding site definition. There are many other optional settings which can be passed to the docking process, but these are essential. Let us set them up:

```
>>> docker = Docker()   # doctest: +SKIP
>>> settings = docker.settings  # doctest: +SKIP
>>> settings.add_protein_file(find_file('1fax_protein.mol2'))   # doctest: +SKIP
```

We can inspect the protein files for the docking:

The easiest way to set up a protein-ligand docking is to read in a previously prepared docking configuration file (e.g. example.conf). These files can be used to run a docking directly, or amended through the API before launching a docking process.

```
>>> conf_file = find_file('example.conf')   # doctest: +SKIP
>>> settings = Docker.Settings.from_file(conf_file) # doctest: +SKIP
>>> docker = Docker(settings=settings)  # doctest: +SKIP
```

If the settings contains all the right information the docking may be launched immediately. In this case, the protein and the ligand files have relative paths, so will need to be changed, and for speed of running this test we will scale the search to 10% of the normal run:

```
>>> settings.clear_protein_files()  # doctest: +SKIP
>>> settings.add_protein_file(find_file('1fax_protein.mol2'))   # doctest: +SKIP
>>> settings.clear_ligand_files()   # doctest: +SKIP
>>> settings.add_ligand_file(find_file('aspirin.mol2'), 10) # doctest: +SKIP
>>> settings.autoscale = 10 # doctest: +SKIP
>>> return_code = docker.dock() # doctest: +SKIP
>>> print(return_code)  # doctest: +SKIP
0
```

However it is possible to set many of the parameters directly. For example, set the files for the docking:

```
>>> settings.ligand_files # doctest: +ELLIPSIS +SKIP
((u'/.../aspirin.mol2', 10),)
```

Note that all files if given a relative path will be located relative to the location of the conf file (which may be set when docking). A ligand file is associated with the number of docking attempts to be tried for each ligand.

Miscellaneous settings

Early termination options may be inspected and set. By default they will be set active, with number of ligands set to 3 and RMSD set to 1.5:

```
>>> settings = Docker().settings # doctest: +SKIP
>>> print(settings.early_termination) # doctest: +SKIP
(True, 3, 1.5)
>>> settings.early_termination = False # doctest: +SKIP
>>> print(settings.early_termination) # doctest: +SKIP
(False, None, None)
>>> settings.early_termination = (True, 5, 2.5) # doctest: +SKIP
>>> print(settings.early_termination) # doctest: +SKIP
(True, 5, 2.5)
```

When setting early termination, number of ligands and RMSD will assume their default values if not specified.

Diverse solutions may be requested for a docking. By default this option is not enabled. If not specified, diverse solution cluster size and RMSD will be given default values.

```
>>> print(settings.diverse_solutions) # doctest: +SKIP
(False, None, None)
>>> settings.diverse_solutions = True # doctest: +SKIP
>>> print(settings.diverse_solutions) # doctest: +SKIP
(True, 1, 1.5)
>>> settings.diverse_solutions = (True, 5, 2.5) # doctest: +SKIP
>>> print(settings.diverse_solutions) # doctest: +SKIP
(True, 5, 2.5)
```

Molecular geometry analysis

Introduction

The CSD-System contains a knowledge-base of intramolecular geometries. This knowledge-base provides easy and rapid access to millions of chemically classified bond lengths, valence angles, acyclic torsion angles, and ring conformations derived from the CSD.

This enables you to rapidly validate the complete geometry of a given query structure and identify any unusual features without the need to construct complex search queries, or carry out detailed data analyses.

Let us import in the **ccdc.conformer** module.

```
>>> from ccdc import conformer
```

Let us also import the **ccdc.io** module so that we can read in molecules of interest.

```
>>> from ccdc import io
```

Note

For more information on these knowledge-based geometry libraries please see: "Retrieval of Crystallographically-Derived Molecular Geometry Information", I. J. Bruno, J. C. Cole, M. Kessler, Jie Luo, W. D. S. Motherwell, L. H. Purkis, B. R. Smith, R. Taylor, R. I. Cooper, S. E. Harris and A. G. Orpen, J. Chem. Inf. Comput. Sci., 44, 2133-2144, 2004 DOI: 10.1021/ci049780b.

Seealso

API documentation of the geometry analyser module

The molecular geometry analysis engine

In order to be able to carry out a molecular geometry analysis one requires an instance of the **ccdc.conformer.GeometryAnalyser** class. Let us create one.

```
>>> engine = conformer.GeometryAnalyser()
```

Note

The most important function of an instance of the **ccdc.conformer.GeometryAnalyser** class is the **ccdc.conformer.GeometryAnalyser.analyse_molecule()** function, which when supplied with a molecule returns a geometry analysed molecule.

Geometry analysis settings

The settings used in a geometry analysis are stored in the settings attribute, which is an instance of the **ccdc.conformer.GeometryAnalyser.Settings** class.

To find out what the current settings are one can make use of the **ccdc.conformer.GeometryAnalyser.Settings.summary()** function.

```
>>> print engine.settings.summary() # doctest: +NORMALIZE_WHITESPACE
Generalisation: True
Impose upper limits: False
Filter rfactor: any
Filter heaviest element: Unknown
Filter solvent: include_solvent
```

```
Filter organometallic: all
Type: bond
    Analyse: True
    Classification measure: Z-score
    Classification measure threshold: 2.00
    Min. Obs. Generalised: 15
    Min. Obs. Exact: 15
    Min. Relevance: 0.75
    Few hits threshold: 5
Type: angle
    Analyse: True
    Classification measure: Z-score
    Classification measure threshold: 2.00
    Min. Obs. Generalised: 15
    Min. Obs. Exact: 15
    Min. Relevance: 0.75
    Few hits threshold: 5
Type: torsion
    Analyse: True
    Classification measure: Local density
    Classification measure threshold: 5.00 interval = 10.00
    Min. Obs. Generalised: 40
    Min. Obs. Exact: 40
    Min. Relevance: 0.75
    Few hits threshold: 15
Type: ring
    Analyse: True
    Classification measure: Local density
    Classification measure threshold: 5.00 interval = 10.00
    Min. Obs. Generalised: 15
    Min. Obs. Exact: 15
    Min. Relevance: 0.75
    Few hits threshold: 15
```

It is also possible to find out the value of a particular setting by accessing it directly. For example to check the minimum number observations required for a generalised ring search one would:

```
>>> engine.settings.ring.min_obs_generalised
15
```

It is possible to change any of the settings directly. For example, let us turn off generalisation.

```
>>> engine.settings.generalisation = False
```

For more information on the setting options available have a look at the **`ccdc.conformer.GeometryAnalyser.Settings`** API documentation.

Performing a geometry analysis on a molecule

Let us read in a PDB ligand, from the structure 1hak, that has had hydrogen atoms added to it and its bond types standardised to CSD conventions.

```
>>> mol_reader = io.MoleculeReader(locate_file('1hak-lig.mol2'))
>>> mol = mol_reader[0]
>>> mol_reader.close()
```

We can now analyse the geometry of this molecule.

```
>>> geometry_analysed_mol = engine.analyse_molecule(mol)
```

Note that this returns a molecule that has had a geometry analysis applied to it and has had additional attributes added to it (for example lists of angles and torsions). We can now find out how many unusual bonds, angles, torsion and rings this molecule has.

Analysing the results

Let us now check how many unusual bonds, angles and

```
>>> len([b for b in geometry_analysed_mol.analysed_bonds if b.unusual])
3
>>> len([a for a in geometry_analysed_mol.analysed_angles if a.unusual])
12
>>> len([t for t in geometry_analysed_mol.analysed_torsions if t.unusual])
6
>>> len([r for r in geometry_analysed_mol.analysed_rings if r.unusual])
1
```

If you only want the unusual features that have enough hits use the **`ccdc.conformer.Analysis.enough_hits`** attribute of a feature.

```
>>> len([t for t in geometry_analysed_mol.analysed_torsions if t.unusual and t.enough_hits])
4
```

If you only wanted the unusual features where there were few hits in the CSD you can use the **`ccdc.conformer.GeometryAnalyser.Analysis.few_hits`** attribute.

```
>>> len([t for t in geometry_analysed_mol.analysed_torsions if t.unusual and t.few_hits])
2
```

To find out which torsions are unusual we can loop over them and print out the atom names.

```
>>> for torsion in geometry_analysed_mol.analysed_torsions:
...     if torsion.unusual and torsion.enough_hits:
...         print torsion.atom_labels
...
[u'O23', u'C21', u'C22', u'C24']
[u'C21', u'C22', u'C24', u'N27']
[u'C31', u'C32', u'C43', u'C52']
[u'C33', u'C32', u'C43', u'C52']
```

To illustrate some of the more advanced functionality exposed let us get the first unusual torsion.

```
>>> tors = geometry_analysed_mol.analysed_torsions
>>> unusual_tors= [t for t in tors if t.unusual and t.enough_hits]
>>> t1 = unusual_tors[0]
```

To access the underlying histogram data one can use the histogram function.

```
>>> t1.histogram()  # doctest: +ELLIPSIS
(2, 5, 4, ... 1, 0, 0)
```

The individual CSD structures used in the analysis can be retrieved using the hit function.

```
>>> hit_mols = t1.hit_molecules
```

To write these out to a sdf file one would use a molecule writer.

```
>>> torsion_identifier = '_'.join(torsion.atom_labels)
>>> torsion_filename = os.path.join(tempdir, torsion_identifier + '.sdf')
>>> writer = io.MoleculeWriter(torsion_filename)
```

```
>>> for molecule in hit_mols:
...     writer.write(molecule)
>>> writer.close()
```

Now let us look at the angles where there are no hits.

```
>>> no_hits = [(i, a) for i, a in enumerate(geometry_analysed_mol.analysed_angles) if a.no_hits]
>>> print len(no_hits)
3
>>> for index, hit in no_hits:
...     print index, hit.fragment_label
14 C10_N11_C21
16 C13_C12_N11
17 C12_C13_S14
```

Let us see if we can find some data for them by using generalisation.

```
>>> engine.settings.generalisation = True
```

In order to speed up the analysis we will only analyse angles. To do this we turn the bond, torsion and ring analysis off.

```
>>> engine.settings.bond.analyse = False
>>> engine.settings.torsion.analyse = False
>>> engine.settings.ring.analyse = False
```

Finally, we do the analysis of the original molecule.

```
>>> analysed_angles = engine.analyse_molecule(mol)
```

We now have data for all the angles.

```
>>> print len([a for a in analysed_angles.analysed_angles if a.no_hits])
0
```

To check the classification of one of the angles for which we previously did not have any hits we can make use of the index in the `no_hits` tuple we created earlier.

```
>>> for index, hit in no_hits:
...     new_hit = analysed_angles.analysed_angles[index]
...     print new_hit.fragment_label, new_hit.classification
C10_N11_C21 Not unusual (enough hits)
C13_C12_N11 Unusual (enough hits)
C12_C13_S14 Not unusual (enough hits)
```

We can determine which angles required generalisation and which did not by inspecting the angles' generalised attribute:

```
>>> print analysed_angles.analysed_angles[6].generalised
True
>>> print analysed_angles.analysed_angles[7].generalised
False
```

It is possible to analyse the geometry of molecules with no 3D information, for example AACFAZ in the CSD:

```
>>> aacfaz = io.MoleculeReader('csd').molecule('AACFAZ')
>>> engine.settings.bond.analyse = True
>>> engine.settings.angle.analyse = True
>>> engine.settings.torsion.analyse = True
>>> engine.settings.ring.analyse = True
>>> checked = engine.analyse_molecule(aacfaz)
```

The checked results will have no values, no classifications, hence no z-score or local density information. There will be no ring information. Checked results will have data associated with the distributions, such as mean, standard_deviation, histograms etc.

```
>>> print checked.analysed_bonds[0].classification
(enough hits)
>>> print checked.analysed_bonds[0].value
None
>>> print round(checked.analysed_bonds[0].mean, 3)
1.511
>>> print checked.analysed_rings
[]
```

Molecular geometry analysis with multiple data libraries

It is now possible to perform a molecular geometry analysis using multiple data libraries simultaneously.

To set up a geometry analysis engine with multiple data libraries the molecular geometry analysis engine should be instantiated with a list of `mogul.path` file names. The path below points to a small example data library.

```
>>> small_geometry_library = 'Small_mogul/mogul.path'
```

Note that the distinguished name `CSD` is used for the molecular geometry library included in the CSD-System.

To set up a molecular geometry analysis engine that made use of the CSD as well as the small example library one would use the syntax below.

```
>>> engine = conformer.GeometryAnalyser(databases=['CSD', small_geometry_library])
```

This molecular geometry analysis engine can be used to analyse molecules as per usual.

```
>>> csd_molecule_reader = io.MoleculeReader('CSD')
>>> mol = csd_molecule_reader.molecule('AWEGOY01')
>>> analysed_mol = engine.analyse_molecule(mol)
```

Hits from the analysis can be drawn from either library. Molecules will be accessed from the corresponding crystal database, either the CSD or the database from which the small example library data files were constructed. The source crystal database may be identified from the hits.

```
>>> hits = analysed_mol.analysed_torsions[0].hits
>>> set([h.source_name for h in hits])
set([u'CSD', u'Small_mogul'])
>>> small_geometry_library_hits = [h for h in hits if h.source_name == 'Small_mogul']
>>> first_small_geometry_library_hit = small_geometry_library_hits[0]
>>> print first_small_geometry_library_hit.molecule.identifier
AWEGOY01
```

Note

The location of the crystal database used to retrieve the crystal structure for a particular hit is specified in the `mogul.path` file. The content of the `mogul.path` file will look something along the lines of:

```
Mogul Data
Date    : 21 02 2013
Version : Unknown
CSD     : Small.ss
Name    : NoSourceDatabase
```

where `Small.ss` is the relative path to the ASER database in question.

It is possible to use a molecular geometry library with no associated ASER database by providing a mogul.path file with no entry for `CSD`. For example:

```
Mogul Data
Date    : 21 02 2013
Version : Unknown
CSD     :
Name    : NoSourceDatabase
```

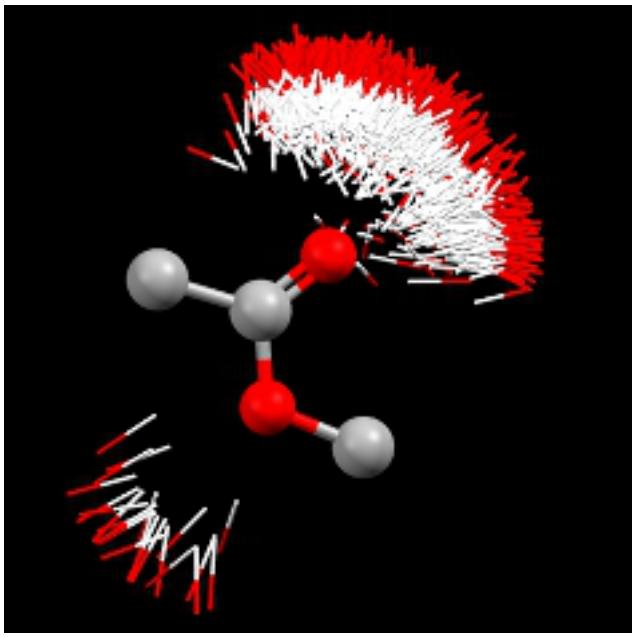
In this case no molecule can be retrieved and ring analysis is not possible. This feature should be considered experimental.

Analysing molecular interactions preferences

Introduction

The CSD-System contains a knowledge-based library of intermolecular interactions. It contains information on the frequencies and directionality of intermolecular contacts is particularly relevant to medicinal chemists interested in identifying bioisosteric replacements and to molecular modellers engaged in structure-based drug design.

Each interaction is represented by a pair of functional groups. The data for each interaction pair in the knowledge-base has been pre-calculated by searching the CSD for non-bonded interactions between a pair of functional groups A and B. The A...B contacts are transformed so that the A groups are least-squared superimposed. The resulting scatterplot shows the experimentally observed distribution of B (the contact group) around A (the central group).



H-bonded O-H contacts around aliphatic ester central group. The majority of these contacts form to the terminal C=O rather than the etheric C-O-C.

Note

For more information on these knowledge-based interaction libraries please see: "Isostar: A library of information about non-bonded interactions" I. J. Bruno, J. C. Cole, J. P. M. Lommerse, R. S. Rowland, R. Taylor and M. L. Verdonk, J., Comput.-Aided Mol. Des., 11, 525-537, 1997 DOI: 10.1023/A:1007934413448.

Seealso

API documentation of the interaction module

Central and contact groups and libraries

Because the data have been pre-calculated we have libraries of the central and contact groups that have been used to populate the knowledge-base. Let us get access to these central and contact group libraries.

```
>>> from ccdc.interaction import InteractionLibrary
>>> central_lib = InteractionLibrary.CentralGroupLibrary()
>>> contact_lib = InteractionLibrary.ContactGroupLibrary()
```

To illustrate the functionality of a group from such a library let us have a look at the aliphatic ester central and the alcohol hydroxyl contact group.

```
>>> central_ester = central_lib.group_by_name('aliphatic-aliphatic ester')
>>> contact_oh = contact_lib.group_by_name('alcohol OH')
```

These groups have name and substructure query attributes. The latter can be used to set up a **ccdc.search.SubstructureSearch**.

```
>>> central_ester.name
u'aliphatic-aliphatic ester'
>>> central_ester.substructure_query  # doctest: +ELLIPSIS
<ccdc.search.QuerySubstructure object at ...>
```

Accessing the underlying interaction data

To access the underlying interaction data for a pair of functional groups we make use of the **ccdc.interaction.InteractionLibrary.CentralGroup.interaction_data()** function. This function takes a **ccdc.interaction.InteractionLibrary.ContactGroup** as an argument.

```
>>> data = central_ester.interaction_data(contact_oh)
```

The relationship between the contact and the central group is symmetric so we could equally have called the **ccdc.interaction.InteractionLibrary.ContactGroup.interaction_data()** function with a **ccdc.interaction.InteractionLibrary.rCentralGroup**.

```
>>> data = contact_oh.interaction_data(central_ester)
```

The data is an instance of the class **ccdc.interaction.InteractionLibrary.InteractionData** and it has several useful attributes. Let us have a look at some of them.

```
>>> data.central_group_name
u'aliphatic-aliphatic ester'
>>> data.contact_group_name
u'alcohol OH'
>>> data.ncontacts
2395
```

The attribute **ccdc.interaction.InteractionLibrary.InteractionData.ncontacts** is the total number of contacts observed in the interaction data. It is also possible to find out how many contacts are within van der Waals distance.

```
>>> data.nvdw_contacts
1682
```

The distance between the contacts in the interaction data are relative to the van der Waals overlap distance. So a contact with a distance of 0.0 would be at van der Waals distance. With this in mind we are in a position to explore data further.

```
>>> data.min_distance
-1.03
>>> data.max_distance
0.5
>>> data.ncontacts_in_range(-1.03, -.50)
1206
```

It is possible to plot a histogram of illustrating the frequency of contacts in 0.01 Angstroms bins.

```
>>> data.histogram  # doctest: +ELLIPSIS
[1, 2, 0, 3, 1, 3, ..., 20, 16, 19, 15, 18, 8]
```

To get a list of pairs of x, y values for this histogram we can use the minimum and maximum distances.

```
>>> width = data.max_distance - data.min_distance
>>> bin_size = float(width) / len(data.histogram)
>>> xs = [ data.min_distance + i*bin_size
...        for i in range(len(data.histogram)) ]
>>> for x, y in zip(xs, data.histogram):  # doctest: +ELLIPSIS
...     print 'Distance %8.2f  contacts %3i' % (x, y)
...
Distance    -1.03  contacts   1
Distance    -1.02  contacts   2
Distance    -1.01  contacts   0
Distance    -1.00  contacts   3
Distance    -0.99  contacts   1
Distance    -0.98  contacts   3
...
Distance     0.44  contacts  20
Distance     0.45  contacts  16
Distance     0.46  contacts  19
Distance     0.47  contacts  15
Distance     0.48  contacts  18
Distance     0.49  contacts   8
```

The interaction data also contains a relative density value and an associated estimated standard deviation. The relative density is defined as $d_{rel} = d_{short}/d_{long}$, where d_{short} is the density of contacts within the sum of van der Waals radii, V, and d_{long} is the density of contacts between V and V+Tol. The larger d_{rel} is, the greater the tendency for the groups to form short interactions. Typical values for d_{rel} are 2-8 for a strong hydrogen bond (*e.g.*, an H-bond where one or both of the groups is/are charged), 1-2 for an average hydrogen bond, 0.7-1 for a weak H-bond, and 0.3-0.7 for a very weak attractive interaction. An estimated standard deviation for d_{rel} is given in brackets, assuming Poisson statistics.

```
>>> print 'Relative density: %.2f (%.2f)' % data.relative_density
Relative density: 2.36 (0.11)
```

Matching functional groups to molecules

Suppose that we wanted to find out if a central and/or contact group was present in a molecule of Aspirin.

```
>>> from ccdc.io import EntryReader
>>> with EntryReader('CSD') as reader:
...     aspirin_entry = reader.entry('ACSALA')
...
>>> print aspirin_entry.chemical_name
2-acetoxybenzoic acid
```

```
>>> central_ester.search_molecule(aspirin_entry.molecule)
[]
```

In this case we do not get a hit using our aliphatic-aliphatic ester as the ester moiety in Aspirin is connected to an aromatic ring. However, if we use an aliphatic-aromatic ester we do get a hit.

```
>>> al_ar_ester = central_lib.group_by_name('aromatic-aliphatic ester')
>>> al_ar_ester.search_molecule(aspirin_entry.molecule)  # doctest: +ELLIPSIS
[<ccdc.interaction.InteractionLibrary.FunctionalGroupHit object at ...>]
```

The **ccdc.interaction.InteractionLibrary.FunctionalGroupHit** object can be used to find the matching atoms in the input molecule and to identify the group that matched.

```
>>> hit = al_ar_ester.search_molecule(aspirin_entry.molecule)[0]
>>> hit.name
u'aromatic-aliphatic ester'
>>> hit.group  # doctest: +ELLIPSIS
<ccdc.interaction.InteractionLibrary.CentralGroup object at ...>
>>> hit.match_atoms()
[Atom(C1), Atom(C3), Atom(C2), Atom(C9), Atom(C8), Atom(O3), Atom(O4)]
>>> hit.match_atoms(indices=True)
(0, 2, 1, 8, 7, 19, 20)
```

If we wanted to find all central/contact groups in a molecule we can use the **ccdc.interaction.CentralGroupLibrary.search_molecule()** and **ccdc.interaction.ContactGroupLibrary.search_molecule()** functions respectively.

```
>>> hits = contact_lib.search_molecule(aspirin_entry.molecule)
>>> len(hits)
38
```

Crystal packing similarity

> **Note**
>
> The crystal packing similarity feature is available only to CSD-Materials and CSD-Enterprise users.

Introduction

A crystal packing similarity metric may be calculated for very similar crystals. This metric is described in "COMPACK: a program for identifying crystal structure similarity using distances", J.A. Chisholm and S. Motherwell, J. Appl. Cryst. 38, 228–231, 2005. DOI: 10.1107/S0021889804027074.

This is a method to identify similarity in molecular packing environments within crystal structures. The method can be used to determine whether two crystal structures are the same to within specified tolerances and also provides a measure of similarity for structures that do not match exactly, but have common structural features. The relative position and orientation of molecules is captured using interatomic distances. This is a representation of structure that avoids use of space-group and cell information.

Using the Packing Similarity API

Firstly we shall import the :class::*ccdc.crystal.PackingSimilarity* class and instantiate it:

```
>>> from ccdc.crystal import PackingSimilarity # doctest: +SKIP
>>> similarity_engine = PackingSimilarity() # doctest: +SKIP
```

The similarity engine currently has just one significant method, to compare to packing shells.

We shall need a couple of crystals to compare:

```
>>> from ccdc.io import CrystalReader
>>> csd = CrystalReader('csd')
>>> fetsec = csd.crystal('FETSEC')
>>> zerlud = csd.crystal('ZERLUD')
```

By default the packing similarity engine prohibits the comparison of different molecular structures. This restriction can be disabled:

```
>>> similarity_engine.settings.allow_molecular_differences = True # doctest: +SKIP
```

Then the structures can be compared:

```
>>> h = similarity_engine.compare(fetsec, zerlud) # doctest: +SKIP
```

The returned value is an instance of :class::*ccdc.crystal.PackingSimilarity.Comparison*. It supports properties to extract the number of molecules of the packing shell which are matched, the number of molecules in the packing shell and the RMSD of the matched structures:

```
>>> print h.nmatched_molecules # doctest: +SKIP
8
>>> print h.packing_shell_size # doctest: +SKIP
15
>>> print round(h.rmsd, 3) # doctest: +SKIP
0.18
```

The **ccdc.crystal.PackingSimilarity** contains a nested class, **ccdc.crystal.PackingSimilarity.Settings** allowing aspects of the similarity measure to be inspected or modified. For example, the distance tolerance which is measured in percent. Relaxing this tolerance will allow more molecules to match with a concomitant raising of the RMSD:

```
>>> print similarity_engine.settings.distance_tolerance # doctest: +SKIP
0.2
>>> similarity_engine.settings.distance_tolerance = 0.4 # doctest: +SKIP
>>> similarity_engine.settings.angle_tolerance = 40. # doctest: +SKIP
>>> h = similarity_engine.compare(fetsec, zerlud) # doctest: +SKIP
>>> print(h.nmatched_molecules, round(h.rmsd, 3)) # doctest: +SKIP
(10, 1.886)
```

Seealso

Packing similarity examples

Generating 2D diagrams of molecules

Introduction

The CSD Python API includes functionality for generating 2D diagrams of molecules. It is particularly effective at generating useful 2D diagrams of large complicated molecules including metal-organic compounds.

The diagram generator makes use of a molecular optimisation framework to generate the 2D diagrams. This includes a global optimisation step, which has a stochastic element to it. One is therefore not guaranteed to get the same layout from different runs.

Let us import the **ccdc.diagram.DiagramGenerator** from the **ccdc.diagram** module. Let us also import a **ccdc.io.MoleculeReader** to allow us to read in molecules from the CSD.

```
>>> from ccdc.diagram import DiagramGenerator
>>> from ccdc.io import EntryReader
```

Seealso

API documentation of the diagram module

Generating 2D diagrams

To generate a 2D diagram for a molecule we need to create an instance of a **`ccdc.diagram.DiagramGenerator`**.

```
>>> diagram_generator = DiagramGenerator()
```

Let us set the font size to 12 and increase the line width and image size.

```
>>> diagram_generator.settings.font_size = 12
>>> diagram_generator.settings.line_width = 1.6
>>> diagram_generator.settings.image_width = 500
>>> diagram_generator.settings.image_height = 500
```

Now let us now read in a moderately complicated compound.

```
>>> csd_reader = EntryReader('CSD')
>>> mol = csd_reader.molecule('EGEQUD')
```

To generate a PIL (Python Imaging Library) image of this molecule we simply call the **`ccdc.diagram.DiagramGenerator.image()`** function.

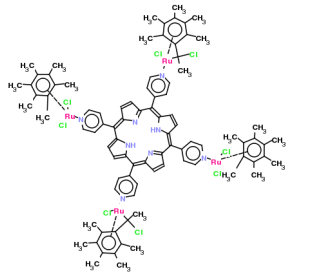
```
>>> img = diagram_generator.image(mol) # img is a PIL (Python Imaging Library) image
```

To save this diagram we can use the PIL image's `save()` function.

Note

More information on PIL can be found in effbot.org/imagingbook/.

Below are diagrams generated for the CSD entries `EGEQUD` a ruthenium porphyrin compound for photodynamic therapy of cancer.



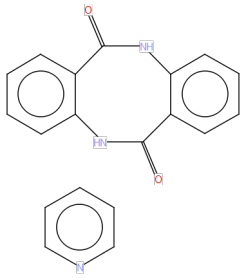
`EGEQUD` 2D diagram; a ruthenium porphyrin compound for photodynamic therapy of cancer.

Accessing diagrams from CSD entries

It is possible to get to the 2D diagram stored in the CSD by giving the **`ccdc.diagram.DiagramGenerator.image()`** an **`ccdc.entry.Entry`**.

```
>>> abebuf = csd_reader.entry('ABEBUF')
>>> img = diagram_generator.image(abebuf)  # img is a PIL (Python Imaging Library) image
```

To save this diagram we can use the PIL image's `save()` function.

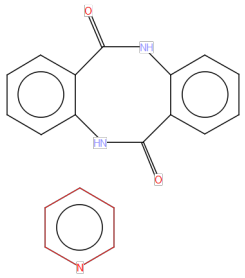


ABEBUF entry 2D diagram.

Highlighting atom selections in diagrams

It is possible to highlight atoms in a diagram. Let us identify the pyridine using a substructure search.

```
>>> from ccdc.search import SubstructureSearch, SMARTSSubstructure
>>> searcher = SubstructureSearch()
>>> sub_id = searcher.add_substructure( SMARTSSubstructure('c1ncccc1') )
>>> hits = searcher.search(abebuf.molecule)
>>> selection = hits[0].match_atoms()
>>> img = diagram_generator.image(abebuf, highlight_atoms=selection)
```



ABEBUF entry 2D diagram with the pyridine ring highlighted.

Warning

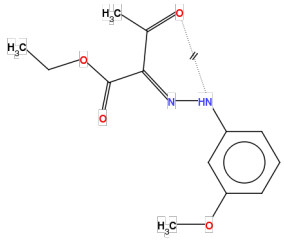
When highlighting selections in **`ccdc.molecule.Molecule`** and **`ccdc.crystal.Crystal`** it is recommended to set the **`ccdc.diagram.DiagramGenerator.Settings.shrink_symbols`** to `False` otherwise it may result in misleading diagrams.

Intra-molecular hydrogen bonds may be displayed. This will require that **`ccdc.diagram.DiagramGenerator.Settings.shrink_symbols`** be set to False. For example:

```
>>> abahui = csd_reader.molecule('ABAHUI')
>>> diagram_generator.settings.detect_intra_hbonds = True
>>> diagram_generator.settings.shrink_symbols = False
```

```
>>> diagram_generator.settings.return_type = 'SVG'
>>> image = diagram_generator.image(abahui)
```

Note that in this case we have set the return type to 'SVG', rather than the default 'PIL'. This is for illustration only; hydrogen bond depiction will work in either format.



ABAHUI with internal H-bond

Diagram generation settings

The **ccdc.diagram.DiagramGenerator** has several settings, which can be modified. These are stored in a **ccdc.diagram.DiagramGenerator.Settings** class.

The default settings are listed below.

```
>>> diagram_settings = DiagramGenerator.Settings()
>>> print diagram_settings.line_width
1.0
>>> print diagram_settings.image_width
350
>>> print diagram_settings.image_height
350
>>> print diagram_settings.font_size  # None: defaults to automatic
None
>>> print diagram_settings.shrink_symbols
True
>>> print diagram_settings.explicit_polar_hydrogens
False
>>> print diagram_settings.detect_intra_hbonds
False
>>> print diagram_settings.overwrite_existing_image
False
>>> print diagram_settings.highlight_color
red
>>> print diagram_settings.return_type
PIL
```

Descriptors

Introduction

The **ccdc.descriptors** module contains functionality for generating geometric, molecular and crystallographic descriptors.

Let us create an **ccdc.io.EntryReader** so that we can read in molecules and crystals from the CSD.

```
>>> from ccdc.io import EntryReader
>>> entry_reader = EntryReader('CSD')
```

As a preamble let us also set up a variable for a temporary directory.

```
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
```

Seealso

API documentation of the descriptors module

Molecular geometry

The module **`ccdc.descriptors.MolecularDescriptors`** contains a number of simple static methods for inspecting the geometry of a molecule. For example,

```
>>> from ccdc.descriptors import MolecularDescriptors as MD, GeometricDescriptors as GD
>>> aabhtz = entry_reader.molecule('AABHTZ')
>>> print round(MD.atom_distance(aabhtz.atom('Cl1'), aabhtz.atom('Cl2')), 2)
5.47
>>> print round(MD.atom_angle(aabhtz.atom('Cl1'), aabhtz.atom('C6'), aabhtz.atom('Cl2')), 2)
178.25
>>> print round(MD.atom_torsion_angle(aabhtz.atom('Cl1'), aabhtz.atom('Cl2'), aabhtz.atom('O2'), aabhtz.atom('O1')), 2)
12.45
```

Note that the atoms do not have to be connected for the geometry to be determined.

One can construct centroids of sets of atoms, vectors from two atoms and RMSD fitted planes. For convenience one can construct centroids and planes directly from a ring:

```
>>> centroid = MD.atom_centroid(*tuple(a for a in aabhtz.atoms))
>>> r1_centroid = MD.ring_centroid(aabhtz.rings[0])
>>> r2_centroid = MD.ring_centroid(aabhtz.rings[1])
>>> r1_plane = MD.ring_plane(aabhtz.rings[0])
>>> r2_plane = MD.ring_plane(aabhtz.rings[1])
```

The plane methods construct instances of **`ccdc.descriptors.GeometricDescriptors.Plane`**. This object supports a few methods for geometric analysis as well as the normal vector and distance from the origin:

```
>>> print round(GD.point_distance(r1_centroid, r2_centroid), 3)
6.385
>>> print round(r1_plane.plane_angle(r2_plane), 3)
88.576
>>> print round(r1_plane.point_distance(r1_centroid), 3)
0.0
>>> print r1_plane.normal
Vector(0.608, -0.050, 0.792)
>>> print round(r1_plane.distance, 3)
-1.785
```

Vectors and planes may be constructed directly from points:

```
>>> x_axis = GD.Vector(1, 0, 0)
>>> vec = GD.Vector.from_points(r1_centroid, centroid)
>>> plane = GD.Plane.from_points(centroid, r1_centroid, r2_centroid)
```

Superimposing molecules

Suppose that we want to superimpose a set of conformers using a substructure of the input molecule rather than all the atoms in it.

This can be achieved using the **`ccdc.MolecularDescriptors.overlay()`** static method.

Let us generate a set of 25 conformers of Rosuvastatin using the PDB ligand from 1hwl as the starting conformation.

```
>>> from ccdc.io import MoleculeReader
>>> filepath = '1hwl-lig.mol2'
```

In this instance we will get the first molecule in the file.

```
>>> mol_reader = MoleculeReader(filepath)
>>> mol = mol_reader[0]
>>> mol_reader.close()
```

Now we set up a conformer generator and generate the 25 conformations.

```
>>> from ccdc.conformer import ConformerGenerator
>>> conformer_generator = ConformerGenerator()  # doctest: +SKIP
>>> conformer_generator.settings.max_conformers = 25  # doctest: +SKIP
>>> conformers = conformer_generator.generate(mol)  # doctest: +SKIP
```

In this instance we want to superimpose the conformers on the pyrimidine ring of the input molecule. We therefore generate a pyrimidine substructure search.

```
>>> from ccdc.search import SubstructureSearch, SMARTSSubstructure
>>> searcher = SubstructureSearch()
>>> sub_id = searcher.add_substructure( SMARTSSubstructure('c1ncncc1') )
```

We can use this to identify the atoms to match in the input molecule.

```
>>> input_hits = searcher.search(mol)
>>> print len(input_hits)
1
>>> input_match_atoms = input_hits[0].match_atoms()
```

Finally, we can use this information to write out a file with conformers superimposed on the pyridine ring of the input structure using the **`ccdc.descriptors.MolecularDescriptors.overlay()`**.

```
>>> from ccdc.descriptors import MolecularDescriptors
>>> from ccdc.io import MoleculeWriter
>>> mol_writer = MoleculeWriter(os.path.join(tempdir, 'superpos.mol2'))
>>> for c in conformers:  # doctest: +SKIP
...     cmol = c.molecule
...     chits = searcher.search(cmol)
...     match_atoms = chits[0].match_atoms()
...     atom_pairs = zip(input_match_atoms, match_atoms)
...     overlaid_mol = MolecularDescriptors.overlay(mol, cmol, atoms=atom_pairs)
...     mol_writer.write(overlaid_mol)
...
```

Another function of interest in this context is the **`ccdc.descriptors.MolecularDescriptors.rmsd`**.

Maximum Common Substructure

`ccdc.descriptors.MolecularDescriptors.MaximumCommonSubstructure` may be used to find a maximum common substructure of two **`ccdc.molecule.Molecule`** instances. We can create an instance of the appropriate class:

```
>>> mcss = MolecularDescriptors.MaximumCommonSubstructure()
```

This instance contains a nested class, **`ccdc.descriptors.MolecularDescriptors.MaximumCommonSubstructure.Settings`** allowing the

configuration of graph search. The meaning of the properties should be evident from their names. Their default settings are as follows:

```
>>> print mcss.settings.ignore_hydrogens
False
>>> print mcss.settings.check_bond_count
False
>>> print mcss.settings.check_bond_polymeric
False
>>> print mcss.settings.check_bond_type
True
>>> print mcss.settings.check_charge
True
>>> print mcss.settings.check_element
True
>>> print mcss.settings.check_hydrogen_count
False
```

Let us read in two molecules with a non-trivial common substructure:

```
>>> csd = MoleculeReader('CSD')
>>> hxacan = csd.molecule('HXACAN')
>>> wexwut = csd.molecule('WEXWUT')
```

We can now determine the maximum common substructure of these two molecules. With default settings only the phenyl ring is common, so we will relax the conditions of the search:

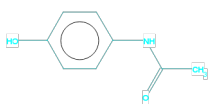
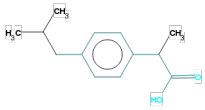
```
>>> mcss.settings.check_element = False
>>> mcss.settings.ignore_hydrogens = True
>>> atoms, bonds = mcss.search(hxacan, wexwut)
```

The atoms returned is a tuple of pairs of atoms matched, the first from HXACAN, the second from WEXWUT. Similarly for matched bonds. We can extract the atoms for either molecule using python's zip function:

```
>>> hxacan_atoms = zip(*atoms)[0]
>>> wexwut_atoms = zip(*atoms)[1]
```

For example, let us make a diagram, highlighting the common substructure in each molecule. Now we can see that with the relaxed search conditions all of HXACAN is matched with a similar group in WEXWUT:

```
>>> from ccdc.diagram import DiagramGenerator
>>> generator = DiagramGenerator()
>>> generator.settings.highlight_color = 'cyan'
>>> generator.settings.shrink_symbols = False
>>> hxacan_image = generator.image(hxacan, highlight_atoms=hxacan_atoms)
>>> wexwut_image = generator.image(wexwut, highlight_atoms=wexwut_atoms)
>>> hxacan_image.save('hxacan_mcss.png')  # doctest: +SKIP
>>> wexwut_image.save('wexwut_mcss.png')  # doctest: +SKIP
```

HXACAN	WEXWUT
	

Seealso

For more information on the available molecular descriptors please have a look at the API documentation: **`ccdc.descriptors.MolecularDescriptors`**.

Powder patterns

Note

The powder pattern features are available only to CSD-Materials and CSD-Enterprise users.

Simulating powder patterns

To illustrate the use of the **`ccdc.descriptors.PowderPattern`** class let us read in a crystal from the CSD.

```
>>> crystal = entry_reader.crystal('AFUSEZ')
```

To create a powder pattern we simply import the **`ccdc.descriptors.PowderPattern`** class and use its **`ccdc.descriptors.PowderPattern.from_crystal()`** static method.

```
>>> from ccdc.descriptors import PowderPattern
>>> pattern = PowderPattern.from_crystal(crystal)
```

The pattern created is an instance of the **`ccdc.descriptors.PowderPattern`** class.

To write the pattern to as a xye or Bruker raw file one can use the functions **`ccdc.descriptors.PowderPattern.write_xye_file()`** and **`ccdc.descriptors.PowderPattern.write_raw_file()`** respectively.

```
>>> pattern.write_xye_file(os.path.join(tempdir, 'afusez.xye'))
```

Comparing powder patterns

It is possible to read in a pattern stored in xye format using the **`ccdc.descriptors.PowderPattern.from_xye_file()`** function.

```
>>> file_pattern = PowderPattern.from_xye_file(os.path.join(tempdir, 'afusez.xye'))
```

Let us determine the similarity between the *AFUSEZ* and *AFUSEA* crystals' calculated powder patterns.

```
>>> afusea_pattern = PowderPattern.from_crystal(entry_reader.crystal('AFUSEA'))
>>> round( file_pattern.similarity(afusea_pattern), 3 )
0.911
```

Settings can be used to control the various parameters of the simulation of a powder pattern. For example to use a non-default wavelength and non-standard two theta range:

```
>>> settings = PowderPattern.Settings()
>>> settings.wavelength = PowderPattern.Wavelength(PowderPattern.Wavelength.Wavelength_FeKa1)
>>> settings.two_theta_minimum = 10.
>>> settings.two_theta_maximum = 70.
>>> pattern = PowderPattern.from_crystal(crystal, settings)
```

Seealso

The *powder pattern cookbook examples*.

Utilities

Introduction

The **ccdc.utilities** module contains a number of general purpose classes.

Seealso

API documentation of the utilities module

Logger

The **ccdc.utilities.Logger** provides a means for the script writer to control output messages. This is a thin wrapper around python's **logging**.

```
>>> from ccdc.utilities import Logger
```

```
>>> logger = Logger()
```

Five categories of message can be emitted: debug, info, warning, error and critical.

```
>>> logger.debug('Generating another conformer.')
```

We do not see anything since the logger's default filter is set at INFO.

```
>>> logger.info('Conformer generation finished.')
INFO <doctest utilities.txt[5]>:1 Conformer generation finished.
```

Because this documentation is tested using doctest, the file and line number are less than relevant. We can turn them off.

```
>>> logger.ignore_line_numbers(True)
>>> logger.warning('Sampling limit reached.')
WARNING Sampling limit reached.
>>> logger.error('Invalid input molecule format.')
ERROR Invalid input molecule format.
>>> logger.critical("The conformer generator failed unexpectedly.")
CRITICAL The conformer generator failed unexpectedly.
```

Messages can be filtered at any level, by default debug messages are filtered out.

```
>>> logger.set_log_level(Logger.DEBUG)
>>> logger.debug('And now we see debug messages.')
DEBUG And now we see debug messages.
```

The class operates as a singleton, *i.e.*, many instances will share the same underlying data.

```
>>> another_logger = Logger()
>>> another_logger.info('Still no line numbers.')
INFO Still no line numbers.
```

Any changes made to either instance will be reflected in the other instance.

```
>>> another_logger.ignore_line_numbers(False)
>>> logger.warning('Logger has line numbers turned back on.')
WARNING <doctest utilities.txt[15]>:1 Logger has line numbers turned back on.
```

Log messages may be redirected to a file, rather than the default stdout.

```
>>> file_name = 'logfile.log'
```

```
>>> another_logger.set_output_file(file_name)
>>> logger.info('This message will go to logfile.log')
```

CCDC code can generate warnings and these will be sent to the logger's destination whether stderr or a file.

CCDC code can issue informative messages. These are be enabled by:

```
>>> from ccdc.io import EntryReader
```

```
>>> logger.set_ccdc_log_level(6)
>>> logger.set_ccdc_minimum_log_level(3)
>>> e = entry_reader[0]
```

This when run in a non-doctest fashion will produce on the logger's output stream:

```
DEBUG 3: AserDatabase::entry extracted: AABHTZ
```

Be warned: setting a *ccdc_minimum_log_level* to 1 will produce a lot of output.

There is one final method of the logger: **`ccdc.utilities.Logger.fatal()`** which will emit a critical error message and exit.

There is a tidy way to redirect messages to a file using a context manager:

```
>> from ccdc.utilities import FileLogger
>> with FileLogger(os.path.join(tempdir, 'redirected.log')) as logger:
...     logger.warning('This message will be in redirected.log')
```

On entry to the block the logger will be set to redirect to a file; on exit the logger will be reset to the default.

Cookbook documentation

Entry examples

Create indexes of useful information for subsets of CSD entries

Note that this script makes use of functionality from the *cookbook utility module*.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
Provide information on a set of structures in the CSD.

This script takes as input a gcd file (a text file with CSD refcodes) and
writes out the identifier, author(s), literature reference, formula, compound
name and compound synonym(s). The output can be formatted as csv or html.

'''
##############################################################################

from __future__ import division, absolute_import, print_function
import sys
import os
import csv
import cgi
import argparse

from ccdc.io import EntryReader

##############################################################################

class Writer(object):
    def __init__(self, infile, out, format='csv'):
        self.rdr = EntryReader(infile, format='identifiers')
        self.out = out
        getattr(self, format + '_header')()
        for e in self.rdr:
            getattr(self, format + '_line')(e)
        getattr(self, format + '_footer')()

    def csv_header(self):
        self.out.writerow([
            'Identifier',
            'Author',
            'Literature Ref',
            'Formula',
            'Compound Name',
            'Synonym'
        ])
```

```
    def csv_footer(self):
        pass

    def csv_line(self, e):
        cit = e.publication
        self.out.writerow([
            e.identifier,
            cit.authors,
            ' '.join(map(str, (cit.journal_name, cit.year, cit.volume, cit.first_page))),
            e.formula,
            e.chemical_name,
            ' '.join(e.synonyms)
        ])

    def html_header(self):
        self.out.write('''
<TABLE border=1>
<TR>
    <TH>Identifier</TH>
    <TH>Author</TH>
    <TH>Literature Ref</TH>
    <TH>Formula</TH>
    <TH>Compound Name</TH>
    <TH>Synonym</TH>
</TR>
''')

    def html_footer(self):
        self.out.write('</TABLE>\n')

    def html_line(self, e):
        cit = e.publication
        self.out.write(
                '<TR><TD>%s</TD><TD>%s</TD><TD>%s</TD><TD>%s</TD><TD>%s</TD><TD>%s</TD></TR>\n' %
            tuple(cgi.escape(x) for x in (
                e.identifier,
                cit.authors,
                ' '.join(map(str, (cit.journal_name, cit.year, cit.volume, cit.first_page))),
                e.formula,
                e.chemical_name,
                ' '.join(e.synonyms)
            ))
        )

##############################################################################

class Arguments(argparse.ArgumentParser):
    '''Options for the program'''
    def __init__(self):
        argparse.ArgumentParser.__init__(self, description=__doc__)
        self.add_argument(
            'input_file',
            help='Location of a gcd file of required refcodes'
        )
        self.add_argument(
            '-o', '--output', default='stdout',
            help='output file [stdout]'
        )
        self.add_argument(
```

```
            '-f', '--format', default='csv', choices=['csv', 'html'],
            help='output format [csv]'
        )
        self.args = self.parse_args()

##############################################################################
```

```
if __name__ == '__main__':
    args = Arguments()
    if args.args.format == 'csv':
        if args.args.output == 'stdout':
            out = csv.writer(sys.stdout)
        else:
            out = csv.writer(open(args.args.output, 'w'))
    else:
        if args.args.output == 'stdout':
            out = sys.stdout
        else:
            out = open(args.args.output, 'w')
    w = Writer(args.args.input_file, out, args.args.format)
    if args.args.format != 'csv':
        out.close()
```

Filter the CSD using a `ccdc.search.Search.Settings` instance

This script will search for given numbers of acceptor atoms and donor atoms or donatable protons. There are arguments to control which CSD entries are acceptable.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
    filter_csd.py - finds molecules in the CSD subject to various criteria

    Only provide the arguments you feel strongly about - the others will have
    defaults which mean they won't have an effect
    Options are -d (or --donors or any prefix), -p or --protons, -a or --acceptors.
    Also -o or --output and -m or --maximum
    Options may be in the form 2,7 to select anything within the range.
    The default for maximum hits is to search the whole CSD.

'''
##############################################################################################################

import sys
import argparse

from ccdc.io import EntryReader, MoleculeWriter
from ccdc.search import Search

def parse_range(s):
    '''Parse and argument into a lower, higher pair.'''
    if s == '':
        bits = [0, 1000]
    elif ',' in s:
        bits = [int(x.strip()) for x in s.split(',')]
    else:
```

```
        bits = [int(s.strip()), int(s.strip())]
    return bits

##############################################################################################################
```

```
class Runner(argparse.ArgumentParser):
    '''Fishes out arguments, runs the search.'''
    def __init__(self):
        super(self.__class__, self).__init__(description=__doc__)
        self.add_argument(
            '-d', '--donors', default='',
            help='number of donor atoms required (may be a range separated by a comma)'
        )
        self.add_argument(
            '-a', '--acceptors', default='',
            help='number of acceptor atoms required (may be a range separated by a comma)'
        )
        self.add_argument(
            '-p', '--protons', default='',
            help='number of donatable protons required (may be a range separated by a comma)'
        )
        self.add_argument(
            '-o', '--output', default='results.gcd',
            help='output file [results.gcd]'
        )
        self.add_argument(
            '-m', '--maximum', default=0, type=int,
            help='Maximum number of structures to find [all]'''
        )

        self.add_argument(
            '-R', '--r-factor', default=5.0, type=float,
            help='Maximum acceptable R-factor [5.0]'
        )
        self.add_argument(
            '-D', '--disorder', default=True, action='store_false',
            help='Whether disordered structures are acceptable [No]'
        )
        self.add_argument(
            '-E', '--errors', default=True, action='store_false',
            help='Whether structures with errors are acceptable [No]'
        )
        self.add_argument(
            '-M', '--organometallic', default=True, action='store_false',
            help='Whether organometallic structures are acceptable [No]'
        )
        self.add_argument(
            '-P', '--polymeric', default=True, action='store_false',
            help='Whether polymeric structures are acceptable [No]'
        )
        self.add_argument(
            '-T', '--two_d', default=True, action='store_false',
            help='Whether 2D structures are acceptable [No]'
        )

        args = self.parse_args()
        if not args.protons and not args.donors and not args.acceptors:
            print "You are asking for everything, and you know a girl can't have it all!"
            sys.exit()

        args.protons = parse_range(args.protons)
        args.donors = parse_range(args.donors)
        args.acceptors = parse_range(args.acceptors)
```

```
        self.args = args
        self.settings = Search.Settings()
        self.settings.only_organic = self.args.organometallic
        self.settings.no_disorder = self.args.disorder
        self.settings.no_errors = self.args.errors
        self.settings.has_3d_coordinates = self.args.two_d
        self.settings.max_r_factor = self.args.r_factor
```

```
        self.settings.max_hit_structures = self.args.maximum # Set this to 0 if you want everything


    def run(self):
        '''Run the search.'''
        csd = EntryReader('csd')
        min_d, max_d = self.args.donors
        min_p, max_p = self.args.protons
        min_a, max_a = self.args.acceptors
        ct = 0

        with MoleculeWriter(self.args.output) as writer:
            for i, e in enumerate(csd):
                if i and i%10000 == 0:
                    print '%d hits from %6d...' % (ct, i)
                if not self.settings.test(e):
                    continue
                try:
                    mol = e.molecule
                except RuntimeError:
                    continue
                donors = [a for a in mol.atoms if a.is_donor]
                acceptors = [a for a in mol.atoms if a.is_acceptor]
                protons = [a for d in donors for a in d.neighbours if a.atomic_number == 1]
                if (
                    (min_d <= len(donors) <= max_d) and
                    (min_p <= len(protons) <= max_p) and
                    (min_a <= len(acceptors) <= max_a)
                    ):
                    writer.write(mol)
                    ct += 1
                    if self.settings.max_hit_structures and ct >= self.settings.max_hit_structures:
                        break

#######################################################################################################################

if __name__ == '__main__':
    r = Runner()
    r.run()
```

Use python's string formatting method to generate an HTML report

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
    simple_report.py - format basic information about a CSD structure in HTML

    This example shows how the python string <a href="https://docs.python.org/2/library/string.html#string-formatting">format method</a>
    may be used to generate an HTML report, with an almost complete
    separation of presentation (the HTML) and logic (the Python source).

    Of course more complicated substitutions may be performed with a
    fully-fledged templating system, such as <a href="www.makotemplates.org">mako</a>,
    <a href="jinja2.pocoo.org">jinja2</a>, but the simplicity of this
    is appealing.
```

```
    Any "{" and "}" in the user's template file, must be replaced with "{{"
    and "}}" for correct substitution.
'''
#######################################################################################

import os
import argparse
```

```
from ccdc.io import EntryReader
from ccdc.diagram import DiagramGenerator

##############################################################################

class Runner(argparse.ArgumentParser):
    '''Reads arguments and writes the report.'''
    def __init__(self):
        '''Defines arguments.'''
        super(self.__class__, self).__init__(description=__doc__)
        self.add_argument(
            'refcodes', nargs='+',
            help='Refcodes for which a report should be generated.'
        )
        self.add_argument(
            '-o', '--output-directory', type=str, default='.',
            help='Directory to which to write the reports [.]'
        )

    def run(self):
        '''Writes a report for each refcode provided.'''
        self.args = self.parse_args()
        self.csd = EntryReader('csd')
        self.generator = DiagramGenerator()
        self.generator.settings.return_type = 'SVG'
        template_file_name = os.path.join(
            os.path.dirname(__file__), 'simple_report_template.html'
        )
        self.template = unicode(open(template_file_name).read())
        for refcode in self.args.refcodes:
            self.run_one(refcode)

    def run_one(self, refcode):
        '''Writes one report.'''
        entry = self.csd.entry(refcode)
        mol = entry.molecule
        atoms = mol.atoms
        bonds = mol.bonds
        img = self.generator.image(mol)
        doi = entry.publication.doi
        if doi is None:
            doi = ' '
        else:
            doi = '<a href="http://dx.doi.org/%s">%s</a>' % (doi, doi)

        with open(os.path.join(self.args.output_directory, refcode + '.html'), 'w') as html:
            s = self.template.format(
                entry=entry,
                molecule=mol,
                image=img,
                doi=doi,
                synonyms='; '.join(s for s in entry.synonyms),
                counts=dict(
                    natoms=len(atoms),
                    ndonors=len([a for a in atoms if a.is_donor]),
                    nacceptors=len([a for a in atoms if a.is_acceptor]),
                    nrot_bonds=len([b for b in bonds if b.is_rotatable]),
                ),
            )
            html.write(s.encode('utf8'))

##############################################################################

if __name__ == '__main__':
    Runner().run()
```

Here is the template used in the script: **`simple_report_template.html`**.

Crystal examples

Generating molecular shells

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#
```

```
"""
Write out heaviest component and molecular shell for a set of CSD structures.

This script takes an input a gcd file and an output directory. For each CSD
entry the script then identifies the heaviest component and writes it to the
output directory. The molecules in the molecular shell around the heaviest
component are then identified and written out to a separate file in the output
directory.

For more help use the -h argument.

"""

from __future__ import division, absolute_import, print_function
import argparse
import os
from ccdc import io

##############################################################################

def main(gcd_file, out_dir):
    """
    Run the main program.
    """
    crystal_reader = io.CrystalReader(gcd_file, format='identifiers')
    for crystal in crystal_reader:

        # Get the atoms from the heaviest component in the crystal. These will
        # be used as the selection around which we create the molecular shell.
        heaviest_component = crystal.molecule.heaviest_component
        selection = heaviest_component.atoms

        # Write out the central molecule
        out_file_name = '%s.mol2' % crystal.molecule.identifier
        out_file_path = os.path.join(out_dir, out_file_name)
        with io.MoleculeWriter(out_file_path) as mol_writer:
            mol_writer.write(heaviest_component)

        # Generate the molecular shell around the atoms in the selection.
        mol_shell = crystal.molecular_shell(distance_type="VdW",
                                            distance_range=(0.0, 0.5),
                                            atom_selection=selection)
        # Write out the molecular shell
        out_file_name = '%s_shell.mol2' % crystal.molecule.identifier
        out_file_path = os.path.join(out_dir, out_file_name)
        with io.MoleculeWriter(out_file_path) as mol_writer:
            mol_writer.write(mol_shell)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('gcd_file', type=str, help="Input GCD file")
    parser.add_argument('out_dir', type=str, help="Output directory")
    args = parser.parse_args()

    if not os.path.isdir(args.out_dir):
        os.mkdir(args.out_dir)
    if not os.path.isfile(args.gcd_file):
        parser.error('%s is not a file.' % args.gcd_file)
```

```
    main(args.gcd_file, args.out_dir)
```

Descriptor HTML report

Note that this script makes use of functionality from the *cookbook utility module*.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

"""
Generate a HTML table of molecular and crystallographic properties for
structures in the CSD.

This includes, amongst more generic ones, point group analysis descriptors.

For more help use the -h argument.

"""

from __future__ import division, absolute_import, print_function
import sys
import argparse
import os.path
import collections

from ccdc import io
from ccdc.molecule import Molecule
from ccdc.crystal import Crystal
from ccdc.descriptors import MolecularDescriptors

this_dir = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(1, this_dir)
import utilities
sys.path.remove(this_dir)

##############################################################################

# The headers for the HTML table
HEADERS = (
    'REFCODE',
    'Identifier',
    'Hydrate',
    'Formula',
    'Space Gp. Symbol',
    'Z Prime',
    'Unique Chem. Units',
    'Mol. Wt.',
    'Donors',
    'Acceptors',
    'Rotatable bonds',
```

```
    'Point Gp. Symbol',
    'Point Gp. Order',
    'Point Gp. Description',
    'Atom counts',
)

##############################################################################

def get_num_unique_structures(molecule):
    "Return the number of unique structures in a molecule."
    d = collections.defaultdict(int)
    for m in molecule.components:
        smi = m.smiles
        if smi is None:
            smi = 'Unknown bond in SMILES'
        d[smi] += 1
    return len(d)

def get_atom_counts(molecule):
    "Return a list of strings of atom counts."
    atom_cts = collections.defaultdict(int)
    for atom in molecule.atoms:
        atom_cts[atom.atomic_symbol] += 1
    l = sorted([(v, k) for k, v in atom_cts.items()])
    l.reverse()
    return ['%s: %d' % (k, v) for v, k in l]

##############################################################################

def get_row_data(entry):
    "Return a tuple of molecular and crystallographic properties."

    # Extract the molecule (which may have several components) and the heavies
    # comonent from the crystal structure. The heaviest component is the
    # molecule of interest.
    crystal = entry.crystal
    mol = crystal.molecule
    mol_of_interest = mol.heaviest_component

    # Find out how many unique molecules are present in the crystal structure
    num_unique_structures = get_num_unique_structures(mol)

    # Ensure all hydrogen atoms are present, for donor calculations
    mol_of_interest.add_hydrogens()

    # Counts of molecular properties
    donors = len([at for at in mol_of_interest.atoms if at.is_donor])
    accs   = len([at for at in mol_of_interest.atoms if at.is_acceptor])
    rots   = len([b  for b  in mol_of_interest.bonds if b.is_rotatable])

    # Collect a list of atoms present (which could be taken from the formula)
    atom_counts = get_atom_counts(mol_of_interest)

    #   Another datum requested
    hydrate = 'hydrate' in entry.chemical_name.lower()

    # Molecular weight (before stripping hydrogen atoms!)
    mol_wt = mol_of_interest.molecular_weight

    #   Strip hydrogens for point group calculations
```

```
    mol_of_interest.remove_hydrogens()

    #   Returns a triple (order, symbol, description)
    pt_group_info = MolecularDescriptors.point_group_analysis(mol_of_interest)

    #   Now we have all the information
    return (
        mol.identifier,
        entry.chemical_name,
        hydrate,
        entry.formula,
        crystal.spacegroup_symbol,
        crystal.z_prime,
        num_unique_structures,
        mol_wt,
        donors,
        accs,
        rots,
        pt_group_info[0],
        pt_group_info[1],
        pt_group_info[2],
        ' '.join(atom_counts)
    )

def main(gcd_file, out_file):
    "Run the main program."

    # Collect the data of interest in a list.
    rows = []
    with io.EntryReader(gcd_file, format='identifiers') as entry_reader:
        for entry in entry_reader:
            rows.append(get_row_data(entry))

    # Write out the report as a HTML file.
    with open(out_file, 'w') as writer:
        writer.write('\n'.join([
            '<HTML>',
            '<HEAD>',
            '<TITLE>Point group analysis</TITLE>',
            '</HEAD>',
            '<BODY>',
            '<H1>Point group analysis</H1>\n'
        ]))
        utilities.write_html_table(HEADERS, rows, stream=writer, border=1)
        writer.write('</BODY></HTML>\n')

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('gcd_file', type=str, help="Input GCD file")
    parser.add_argument('out_file', type=str, help="Output HTML file")
    args = parser.parse_args()

    if not os.path.isfile(args.gcd_file):
        parser.error('%s is not a file.' % args.gcd_file)

    main(args.gcd_file, args.out_file)
```

Molecular processing examples

Standardising, converting and filtering molecules

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#
"""
Filter and transform molecules.

Input specification should precede all other arguments with the exception of
an optional -v.  If input is 'CSD' the current CSD will be
filtered/transformed, if it is 'Xpress' the latest CSD XPress database will be
used.  Otherwise a gcd or mol2 file will be used.

Other filters and transformers will be applied in the order given, though a
single output option will be applied last.

Any filter or transformer may be preceded by -v to enable logging of that
processor.

For more help use the -h argument.

"""

from __future__ import division, absolute_import, print_function
import sys
import os
import glob
import time
import argparse

import logging

if sys.version_info[0] == 2:
    # Make range's behaviour consistent between Python 2 and Python 3
    range = xrange

import ccdc.io
import ccdc.molecule
import ccdc.search

from ccdc.utilities import Logger
from functools import reduce

###############################################################################

class MolProcessor(object):
    """Base class for readers, writers, filters and transformers."""

    def __init__(self, source, **kw):
        """Updates dictionary with keyword arguments."""
        self.source = source
        self.__dict__.update(kw)
        self.successes = 0
```

```
        self.failures = 0
        self.logger = Logger()
        self.logger.ignore_line_numbers()
        self.logger.set_log_level(logging.INFO)

    def __str__(self):
        """Defaults to class name."""
        return self.__class__.__name__
    __repr__ = __str__

    def molecules(self):
        """Generator for molecules. Yields a molecule."""
        for m in self.source.molecules():
            m = self.process(m)
            if m:
                yield m

    def process(self, m):
        """Default to do nothing."""
        return m

    def message(self, *args):
        """Unconditionally print a message."""
        self.logger.info(' '.join(map(str, args)))

    def log(self, *args):
        """Log a message if verbose."""
        if self.verbose:
            self.logger.info(' '.join(map(str, args)))

    def succeed(self, *args):
        """Register a success, and optionally log it."""
        self.successes += 1
        if self.verbose:
            self.logger.info(
                '%s: succeeded %s' % (self, ' '.join(map(str, args))) )

    def fail(self, *args):
        """Register a failure and optionally log it."""
        self.failures += 1
        if self.verbose:
            self.logger.info(
                '%s: failed %s' % (self, ' '.join(map(str, args))) )

    def print_stats(self):
        """Print statistics on success and failure."""
        if self.source is not None:
            self.source.print_stats()
        self.logger.info('%6d total, %6d successes, %6d failures, %s' % (
                self.successes+self.failures,
                self.successes,
                self.failures,
                self
            ))

    @classmethod
    def long_option(klass):
        """Construct a sensible long option name."""
        s = '-'
        for k in klass.__name__:
```

```
            if k in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
                s += '-' + k.lower()
            else:
                s += k
        return s

##############################################################################

class MolSource(MolProcessor):
    """Sources of molecules."""
    def process(self, m):
        raise NotImplementedError('MolSource does not implement process')

    class ParseArgs(argparse.Action):
        def __call__(self, parser, namespace, values, option_string):
            verbose = namespace.verbose or namespace.Verbose
            namespace.verbose = False
            fname = values[0]
            if '*' in fname:
                s = GlobSource(fname, verbose=verbose)
            else:
                s = ccdc.io.MoleculeReader(fname)
            if namespace.source is None:
                namespace.source = s
            else:
                namespace.source = CompoundSource(namespace.source,
                                                  s,
                                                  verbose=verbose)

    def __str__(self):
        return 'MoleculeReader(%s)' % self.file_name

class CompoundSource(MolSource):
    """All molecules from two sources."""
    def __init__(self, first, second, verbose=False):
        MolProcessor.__init__(self, None)
        self.first = first
        self.second = second
        self.verbose = verbose

    def molecules(self):
        """Iterates over first, then second."""
        for m in self.first.molecules():
            yield m
        for m in self.second.molecules():
            yield m

    def __str__(self):
        return 'Compound(%s, %s)' % (self.first, self.second)

class GlobSource(MolSource):
    """Read all molecules from all files matched by the globbing pattern."""
    def __init__(self, pattern, verbose=False):
        """Get all files"""
        MolProcessor.__init__(self, None, verbose=verbose)
        self.file_name = pattern
        self.files = glob.glob(pattern)
        self.files.sort()

    def molecules(self):
```

```
        """Iterate over all molecules of all files."""
        for f in self.files:
            with ccdc.io.MoleculeReader(f) as source:
                for i, m in enumerate(source):
                    if i > 0:
                        m.identifier = '%s_%d' % (
                            os.path.splitext(os.path.basename(f))[0], i )
                    else:
                        m.identifier = os.path.splitext(os.path.basename(f))[0]
                    self.succeed(m.identifier)
                    yield m

##############################################################################

class MolSink(MolProcessor):
    """Output files."""
    def __init__(self, source, fname=None, verbose=False, format=''):
        """Opens a stream as well as storing the file name."""
        MolProcessor.__init__(self, source, fname=fname, verbose=verbose)
        self.sink = ccdc.io.MoleculeWriter(fname, format=format)

    def __str__(self):
        """Report file name if any."""
        return 'MoleculeWriter(%s)' % self.sink.file_name

    def molecules(self):
        """Doesn't produce any molecules."""
        raise StopIteration

    def close(self):
        """Close the sink."""
        self.sink.close()

    def run(self, limit=sys.maxsize):
        """This pulls the chain of processors."""
        ct = 0
        for m in self.source.molecules():
            if ct >= limit:
                break
            self.sink.write(m)
            ct += 1

class GlobSink(MolSink):
    "Write molecules to separate files based on their identifier and the format."
    def __init__(self, source, format='', verbose=False):
        """Save the format"""
        MolProcessor.__init__(self, source, verbose=verbose)
        self.format = format

    def __str__(self):
        """Show the name."""
        return 'MoleculeWriter(%s)' % self.format

    def close(self):
        """nothing to close"""
        pass

    def run(self, limit=sys.maxsize):
        """Writes a molecule to a file based on pattern and filename.
        Will create all necessary directories.
```

```
        """
        ct = 0
        for m in self.source.molecules():
            if ct >= limit:
                break
            fname = self.format.replace('*', m.identifier)
            d = os.path.dirname(fname)
            if d and not os.path.isdir(d):
                os.makedirs(d)
            with ccdc.io.MoleculeWriter(fname) as sink:
                sink.write(m)
                self.succeed(m.identifier)
            ct += 1

class CanonicalSmiles(MolSink):
    """Write canonical SMILES."""
    def __init__(self, source, fname=None, verbose=False):
        """Sets up generators for the calculation"""
        MolProcessor.__init__(self, source, verbose=verbose)
        if fname:
            self.stream = open(fname, 'w')
        else:
            self.stream = sys.stdout

    def generate(self, m):
        """Generate the SMILES."""
        return m.smiles

    def run(self, limit=sys.maxsize):
        ct = 0
        for m in self.source.molecules():
            if ct >= limit:
                break
            s = self.generate(m)
            if s is None:
                self.stream.write(s + '\n')
                self.succeed(m.identifier)
                ct += 1
            else:
                self.fail(m.identifier, 'Cannot calculate SMILES')

class UniqueSmiles(CanonicalSmiles):
    """Write canonical smiles without duplication."""
    def __init__(self, source, fname=None, verbose=False):
        CanonicalSmiles.__init__(self, source, fname=fname, verbose=verbose)
        self.seen = collections.defaultdict(list)

    def process(self, m):
        s = m.smiles
        if s is None:
            self.fail('Cannot calculate SMILES')
        else:
            if s in self.seen:
                self.fail(m.identifier, 'Duplicate')
            else:
                self.succeed(m.identifier, s)
                self.stream.write(s + '\n')
            self.seen[s].append(m.identifier)

    def print_stats(self):
```

```
        MolProcessor.print_stats(self)
        self.logger.info('%6d Unique out of %6d structures' % (
            len(self.seen),
            len(reduce(list.__add__, self.seen.values(), []))
        ))


################################################################################

class NullaryMolProcessor(MolProcessor):
    """Processors requiring no arguments."""
    class Callback(argparse.Action):
        def __call__(self, parser, namespace, values, option_string=None):
            """Set the verbose flag and construct the class."""
            verbose = namespace.verbose or namespace.Verbose
            namespace.source = self.klass(namespace.source, verbose=verbose)
            namespace.verbose = False

    @classmethod
    def add_option(klass, group):
        group.add_argument(
            klass.long_option(),
            help=klass.__doc__,
            nargs=0,
            action=NullaryMolProcessor.Callback
        ).klass = klass

class UnaryMolProcessor(MolProcessor):
    class Callback(argparse.Action):
        def __call__(self, parser, namespace, values, option_string=None):
            verbose = namespace.verbose or namespace.Verbose
            namespace.verbose = False
            if self.klass.validate_arg(values[0]):
                namespace.source = self.klass(namespace.source,
                                              arg=values[0],
                                              verbose=verbose)
            else:
                raise argparse.ArgumentError(self,
                    '%s %s invalid argument %s' % (
                        self.klass.__name__,
                        self.klass.__doc__,
                        ' '.join(values)
                    )
                )

    @classmethod
    def add_option(klass, group):
        group.add_argument(
            klass.long_option(), help=klass.__doc__,
            action=klass.Callback, nargs=1
        ).klass = klass

    @classmethod
    def validate_arg(klass, value):
        return True

class IntRangeMolProcessor(UnaryMolProcessor):
    """Processors requiring a pair of integers."""
    class Callback(argparse.Action):
        def __call__(self, parser, namespace, values, option_string):
```

```
            """Extract arguments, set verbosity and construct the class."""
            value = values[0]
            verbose = namespace.verbose or namespace.Verbose
            namespace.verbose = False
            try:
                l, h = [int(x) for x in value.split(',')]
            except (TypeError, ValueError):
                try:
                    l, h = (int(x) for x in value.split(':'))
                except (TypeError, ValueError):
                    raise argparse.ArgumentError(
                        self,
                        '%s requires an int,int pair as argument' % option_string
                    )
            namespace.source = self.klass(
                namespace.source, low=l, high=h, verbose=verbose
            )

    def __str__(self):
        """Report the range."""
        return '%s (%d, %d)' % (self.__class__.__name__, self.low, self.high)

class FloatRangeMolProcessor(UnaryMolProcessor):
    """Processors requiring two real values."""
    class Callback(argparse.Action):
        def __call__(self, parser, namespace, values, option_string):
            """Extract arguments, set verbosity and construct the class."""
            value = values[0]
            try:
                l, h = (float(x) for x in value.split(','))
            except (TypeError, ValueError):
                try:
                    l, h = (float(x) for x in value.split(':'))
                except (TypeError, ValueError):
                    raise argparse.ArgumentError(
                        self,
                        '%s requires a float,float pair as argument' % option_string
                    )
            verbose = namespace.verbose or namespace.Verbose
            namespace.verbose = False
            namespace.source = self.klass(namespace.source,
                                          low=l,
                                          high=h,
                                          verbose=verbose )

    def __str__(self):
        """Report the range."""
        return '%s (%.2f, %.2f)' % (self.__class__.__name__,
                                    self.low, self.high )

class FloatMolProcessor(UnaryMolProcessor):
    class Callback(argparse.Action):
        def __call__(self, parser, namespace, values, option_string):
            "Extract single argument, set verbosity and construct the class."
            value = values[0]
            try:
                v = float(value)
            except TypeError:
                raise argparse.ArgumentError(
                    self,
```

```
                    '%s requires a float value as argument' % option_string
                )
            namespace.source = self.klass(
                namespace.source, verbose=verbose, **{klass.variable: v}
            )

    def __str__(self):
        """Report the value."""
        return '%s (%s=%.2f)' % (self.__class__.__name__,
                                 self.variable,
                                 getattr(self, self.variable))

##############################################################################
#   Transformers
##############################################################################

class LargestComponent(NullaryMolProcessor):
    """Extract the heaviest component from a multi-component structure."""
    def process(self, m):
        l = sorted([(x.molecular_weight, x) for x in m.components])
        self.succeed(m.identifier,
                     '%d components, heaviest %.2f' % (len(l), l[-1][0]) )
        return l[-1][1]

class AddHydrogens(NullaryMolProcessor):
    """Add hydrogen atoms to a molecule.

    If we want sites, we need a crystal structure or a cell.
    """
    def process(self, m):
        nats = len(m.atoms)
        m.add_hydrogens()
        self.succeed(m.identifier, 'before', nats, 'now', len(m.atoms))
        return m

class RemoveHydrogens(NullaryMolProcessor):
    """Remove hydrogen atoms from a molecule."""
    def process(self, m):
        nats = len(m.atoms)
        m.remove_hydrogens()
        self.succeed(m.identifier, 'before', nats, 'now', len(m.atoms))
        return m

class NormaliseHydrogens(NullaryMolProcessor):
    """Normalise the positions of hydrogen atoms in a molecule."""
    def process(self, m):
        m.normalise_hydrogens()
        self.succeed(m.identifier, 'normalised')
        return m

class RemoveWaters(NullaryMolProcessor):
    """Remove water molecules from entries."""
    def process(self, m):
        keep = []
        waters = 0
        for s in m.components:
            ats = [at.atomic_symbol for at in s.atoms]
            if len(ats) == 3:
                ats.sort()
                if ats[0] == 'H' and ats[1] == 'H' and ats[2] == 'O':
```

```
                    waters += 1
                else:
                    keep.append(s)
            else:
                keep.append(s)
        new = ccdc.molecule.Molecule()
        for k in keep:
            new.add_molecule(k)
        self.succeed(m.identifier,
                     waters,
                     'removed',
                     len(keep),
                     'structures retained')
        return m

class Neutralise(NullaryMolProcessor):
    """Remove formal charge from all atoms."""
    def process(self, m):
        ct = 0
        for a in m.atoms:
            if a.formal_charge != 0:
                a.formal_charge = 0
                ct += 1
        self.succeed(m.identifier, '%d atoms neutralised' % ct)
        return m

class StandardiseAromaticBonds(NullaryMolProcessor):
    """Standardise aromatic bonds to CSD conventions."""
    def process(self, m):
        bond_map = [b.bond_type for b in m.bonds]
        m.standardise_aromatic_bonds()
        bonds_now = [b.bond_type for b in m.bonds]
        diffs = sum(a != b for a, b in zip(bond_map, bonds_now))
        self.succeed(m.identifier, 'changed %d bonds' % diffs)
        return m

class StandardiseDelocalisedBonds(NullaryMolProcessor):
    """Standardise delocalised bonds to CSD conventions."""
    def process(self, m):
        bond_map = [b.bond_type for b in m.bonds]
        m.standardise_delocalised_bonds()
        bonds_now = [b.bond_type for b in m.bonds]
        diffs = sum(a != b for a, b in zip(bond_map, bonds_now))
        self.succeed(m.identifier, 'changed %d bonds' % diffs)
        return m

class AssignBondTypes(UnaryMolProcessor):
    """Assign bond types. Either 'all' or 'unknown'."""
    def process(self, m):
        bond_map = [b.bond_type for b in m.bonds]
        m.assign_bond_types(self.arg)
        bonds_now = [b.bond_type for b in m.bonds]
        diffs = sum(a != b for a, b in zip(bond_map, bonds_now))
        self.succeed(m.identifier, 'changed %d bonds' % diffs)
        return m

    @classmethod
    def validate_arg(klass, value):
        return value in ['all', 'unknown']
```

```
################################################################################
#   Filters
################################################################################

class MolecularWeight(FloatRangeMolProcessor):
    """Filters by molecular weight: float,float."""
    def process(self, m):
        if self.low <= m.molecular_weight <= self.high:
            self.succeed(m.identifier, m.molecular_weight)
        else:
            self.fail(m.identifier, m.molecular_weight)
        return m

class Organic(NullaryMolProcessor):
    """Permit only organic structures."""
    def process(self, m):
        if m.is_organic:
            self.succeed(m.identifier, 'organic')
        else:
            self.fail(m.identifier, 'inorganic or organometallic')
        return m

class AtomCount(IntRangeMolProcessor):
    """Filter by all atom count: int,int."""
    def process(self, m):
        if self.low <= len(m.atoms) <= self.high:
            self.succeed(m.identifier, '%d atoms' % len(m.atoms))
            return m
        else:
            self.fail(m.identifier, '%d atoms' % len(m.atoms))

class RotatableBondCount(IntRangeMolProcessor):
    """Filter by rotatable bond count: int,int."""
    def process(self, m):
        c = sum(b.is_rotatable for b in m.bonds)
        if self.low <= c <= self.high:
            self.succeed(m.identifier, '%d rotatable bonds' % c)
            return m
        else:
            self.fail(m.identifier, '%d rotatable bonds' % c)

class DonorCount(IntRangeMolProcessor):
    """Filter by HBond donor count: int,int."""
    def process(self, m):
        c = sum(a.is_donor for a in m.atoms)
        if self.low <= c <= self.high:
            self.succeed(m.identifier, '%d donor atoms' % c)
            return m
        else:
            self.fail(m.identifier, '%d donor atoms' % c)

class AcceptorCount(IntRangeMolProcessor):
    """Filters by HBond acceptor count: int,int."""
    def process(self, m):
        c = sum(a.is_acceptor for a in m.atoms)
        if self.low <= c <= self.high:
            self.succeed(m.identifier, '%d acceptors' % c)
            return m
        else:
            self.fail(m.identifier, '%d acceptors' % c)
```

```
class FormalCharge(IntRangeMolProcessor):
    """Filter by formal charge: int,int."""
    def process(self, m):
        c = m.formal_charge
        if self.low <= c <= self.high:
            self.succeed(m.identifier, '%d formal charge' % c)
            return m
        else:
            self.fail(m.identifier, '%d formal charge' % c)

class MetalCount(IntRangeMolProcessor):
    """Filter by number of metals: int, int."""
    def process(self, m):
        c = sum(a.is_metal for a in m.atoms)
        if self.low <= c <= self.high:
            self.succeed(m.identifier, '%d metals' % c)
            return m
        else:
            self.fail(m.identifier, '%d metals' % c)

class SameSiteMetals(NullaryMolProcessor):
    """Filter by whether two metals share the same site."""
    def process(self, m):
        metals = [at for at in m.atoms if at.is_metal]
        for i in range(len(metals)):
            for j in range(i):
                one = metals[i].coordinates
                two = metals[j].coordinates
                if ( one.x() == two.x()
                     and one.y() == two.y()
                     and one.z() == two.z() ):
                    self.fail(m.identifier,
                              '%s and %s share a site' % (metals[i],
                                                          metals[j]))
        self.succeed(m.identifier, 'No metals share a site')
        return m

class AllAtomsHaveSites(NullaryMolProcessor):
    """Filter on whether all atoms have sites."""
    def process(self, m):
        a = m.all_atoms_have_sites
        if a:
            self.succeed(m.identifier, 'all atoms have sites')
            return m
        else:
            self.fail(m.identifier, 'some atoms have no sites')

class NumberOfComponents(IntRangeMolProcessor):
    """Filter on count of distinct entities in a structure: int,int."""
    def process(self, m):
        c = len(m.components)
        if self.low <= c <= self.high:
            self.succeed(m.identifier, '%d components' % c)
            return m
        else:
            self.fail(m.identifier, '%d components' % c)

class SmallestRingSize(IntRangeMolProcessor):
    '''Filter on minimum and maximum size of a ring'''
```

```
    def process(self, m):
        if not hasattr(self, 'query'):
            at = ccdc.search.QueryAtom()
            at.smallest_ring = [self.low, self.high]
            self.query = ccdc.search.QuerySubstructure()
            self.query.add_atom(at)
        if self.query.search_molecule(m, limit=1):
            self.succeed(m.identifier, 'smallest ring')
            return m
        else:
            self.fail(m.identifier, 'smallest ring')

class UnfusedUnbridgedRingSize(IntRangeMolProcessor):
    '''Filter for ...'''
    def process(self, m):
        if not hasattr(self, 'query'):
            at = ccdc.search.QueryAtom()
            at.smallest_ring = [self.low, self.high]
            at.unfused_unbridged_ring = True
            self.query = ccdc.search.QuerySubstructure()
            self.query.add_atom(at)
        if self.query.search_molecule(m, limit=1):
            self.succeed(m.identifier, 'unfused ring')
            return m
        else:
            self.fail(m.identifier, 'unfused ring')


##############################################################################

class Arguments:
    """Holds options and is used to build the processor."""
    def __init__(self):
        self.source = None
        self.verbose = False
        self.Verbose = False

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        prog='molprocessor.py',
        description=__doc__
    )
    group = parser.add_argument_group('Files')
    group.add_argument(
        '-i', '--input', help="""\
Input file name or CSD to process the whole of the CSD. An input should be at
the start of a chain of processors, but additional inputs may appear at any
other point to allow pre-processed files to be appended.""",
        action=MolSource.ParseArgs, nargs=1
    )

    group.add_argument(
        '-o', '--output', help='Output file'
    )

    group = parser.add_argument_group('Filters')
    MolecularWeight.add_option(group)
    AtomCount.add_option(group)
    FormalCharge.add_option(group)
    AllAtomsHaveSites.add_option(group)
```

```
    RotatableBondCount.add_option(group)
    DonorCount.add_option(group)
    AcceptorCount.add_option(group)
    NumberOfComponents.add_option(group)
    MetalCount.add_option(group)
    SameSiteMetals.add_option(group)
    Organic.add_option(group)
    SmallestRingSize.add_option(group)
    UnfusedUnbridgedRingSize.add_option(group)

    group = parser.add_argument_group('Transformers')
    LargestComponent.add_option(group)
    AddHydrogens.add_option(group)
    RemoveHydrogens.add_option(group)
    NormaliseHydrogens.add_option(group)
    RemoveWaters.add_option(group)
    Neutralise.add_option(group)
    StandardiseAromaticBonds.add_option(group)
    StandardiseDelocalisedBonds.add_option(group)
    AssignBondTypes.add_option(group)

    group = parser.add_argument_group('Miscellaneous')
    group.add_argument(
        '-v', '--verbose',
        help='Sets the following processor to print as it processes',
        action='store_true',
        default=False
    )
    group.add_argument(
        '-V', '--Verbose',
        help='Sets all processors to verbose',
        action='store_true',
        default=False
    )
    group.add_argument(
        '-s', '--silent',
        help='Do not print statistics at the end',
        action='store_true',
        default=False
    )
    group.add_argument(
        '-l', '--limit',
        help='Limit the number of structures written',
        type=int,
        default=sys.maxsize
    )

    arguments = Arguments()
    parser.parse_args(sys.argv[1:], namespace=arguments)

    if arguments.output:
        if arguments.output.endswith('.smiles'):
            sink = CanonicalSmiles(arguments.source, arguments.output)
        elif '*' in arguments.output:
            sink = GlobSink(arguments.source, arguments.output)
        else:
            sink = MolSink(arguments.source, arguments.output)
    else:
        sink = MolSink(arguments.source, 'stdout', format='identifiers')
```

```
    sink.run(limit=int(arguments.limit))
    sink.close()
    sys.exit()
    if not arguments.silent:
        sink.print_stats()
```

Search examples

Torsion angle preferences in drug-like molecules

Note that this script makes use of functionality from the *cookbook utility module*.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
Script to generate torsion angle histograms for common med. chem. fragments.

This script will create a report containing torsion histograms for ten common
medicinal chemistry fragments in an output directory.

The script was inspired by and the SMARTS pattern taken from the supplementary
material of:

"Torsion Angle Preferences in Druglike Chemical Space:
A Comprehensive Guide"
Scharfer C.; Schulz-Gasch T.; Ehrilich H.-C.; Guba W.; Rarey M.; Stahl M.,
J. Med. Chem. (2013) 56, 2016-2028

'''

##############################################################################

from __future__ import division, absolute_import, print_function
import sys
import os
import argparse
import time

from ccdc.search import SMARTSSubstructure, SubstructureSearch
from .utilities import Histogram

##############################################################################

#   Define arguments and parse them

def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('-m', '--max-hits', type=int,
                        default=sys.maxsize,
                        help='Maximum number of entries to use'
```

```
    )

    parser.add_argument('-o', '--out-dir',
                        default='med_chem_torsions',
                        help='Output directory [med_chem_torsions]'
    )
    args = parser.parse_args()

#   Extract file names and create directory if necessary
    if not os.path.exists(args.out_dir):
        os.mkdir(args.out_dir)

##############################################################################

#   Top ten SMARTS patterns

    patts = [
        '[*:1]~[CX3:2]!@[NX3:3]~[*:4]',
        '[*:1]~[NX3:2]!@[NX2:3]~[*:4]',
        '[*:1]~[NX3:2]!@[NX3:3]~[*:4]',
        '[cH1:1]~[a:2]([cH1])!@[a:3]([s,o,n:4])([cH0])',
        '[*:1]~[cX3:2]!@[NX2:3]~[*:4]',
        '[*:1]~[CX4:2]!@[NX2:3]~[*:4]',
        '[cH1:1]~[a:2]([cH1])!@[a:3]([cH0])[cH0:4]',
        '[*:1]~[OX2:2]!@[P:3]~[*:4]',
        '[*:1]~[CX4:2]!@[P:3]~[*:4]',
        '[*:1]~[CX4:2]!@[CX3:3]~[*:4]',
    ]

##############################################################################

#   Do the search, and draw the histograms

    hist_filenames = []
    for i, p in enumerate(patts):
        t = time.time()
        q = SMARTSSubstructure(p)
        s = SubstructureSearch()
        s.add_substructure(q)
        s.add_torsion_angle_measurement('Torsion',
            0, q.label_to_atom_index(1),
            0, q.label_to_atom_index(2),
            0, q.label_to_atom_index(3),
            0, q.label_to_atom_index(4)
        )
        hits = s.search(max_hit_structures=args.max_hits,
                        max_hits_per_structure=1)
        hist_fn = os.path.join(args.out_dir, 'torsion_%02d.png' % i)
        hist_filenames.append(hist_fn)
        hist = Histogram(title=p, xlabel='Torsion', ylabel='Count',
                         file_name=hist_fn)
        xs = [abs(h.measurements['Torsion']) for h in hits]
        hist.add_plot(xs, color='blue', bins=18)
        hist.write()

##############################################################################

#   Write the html

    f = open(os.path.join(args.out_dir, 'index.html'), 'w')
```

```
    f.write("<html><head><title>Torsion angle preferences</title></head>\n")
    f.write("<body><h1>Druglike chemical space torsion angle preferences</h1>\n")
    for hist_fn in hist_filenames:
        f.write("<img src='%s'/>\n" % os.path.basename(hist_fn))
    f.write("</body></html>\n")
    f.close()

##############################################################################

if __name__ == '__main__':
    main()
```

Similarity search with maximum common substructure highlighted

This example performs a similarity search, then generates diagrams of the matching structures showing the maximum common substructure if the probe molecule and the hit molecule.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
    maximum_common_substructure.py  -   perform a similarity search, then format a web page containing
        the common substructures of the original structure and the returned similar structures.
'''
##############################################################################

import os
import argparse

from ccdc.io import MoleculeReader
from ccdc.search import SimilaritySearch
from ccdc.descriptors import MolecularDescriptors
from ccdc.diagram import DiagramGenerator

##############################################################################

class Runner(argparse.ArgumentParser):
    '''Defines arguments, reads command line, performs the search and formats the report.'''
    def __init__(self):
        '''Defines arguments.'''
        super(self.__class__, self).__init__(description=__doc__)
        self.add_argument(
            'refcodes', nargs='+',
            help='CSD refcodes or files from which to read molecules.'
        )
        self.add_argument(
            '-t', '--threshold', type=float, default=0.9,
            help='Similarity threshold sought'
        )
        self.add_argument(
            '-m', '--maximum', type=int, default=0,
```

```
            help='Maximum number of structures to retrieve [all]'
        )
        self.add_argument(
            '-o', '--output', default='results.html',
            help='Where to write the HTML report [report.html]'
        )
```

```
    def run(self):
        '''Runs the task.'''
        self.args = self.parse_args()
        csd = MoleculeReader('csd')
        self.mcss = MolecularDescriptors.MaximumCommonSubstructure()
        self.generator = DiagramGenerator()
        self.generator.settings.return_type = 'SVG'
        self.generator.settings.highlight_color = '#00C0C0'
        self.generator.settings.shrink_symbols = False
        self.output = open(self.args.output, 'w')
        self.output.write('<html><head><title>Similarity Report</title></head><body>\n')
        for r in self.args.refcodes:
            if os.path.exists(r):
                for m in MoleculeReader(r):
                    self.run_one(m)
            else:
                self.run_one(csd.molecule(r))
        self.output.write('</body></html>\n')
        self.output.close()

    def run_one(self, mol):
        '''Runs one search and report generation.'''
        search = SimilaritySearch(mol, self.args.threshold)
        hits = search.search(max_hit_structures=self.args.maximum)
        def one_hit(hit):
            '''Format a single hit.'''
            atoms, bonds = self.mcss.search(mol, hit.molecule)
            mol_atoms = zip(*atoms)[0]
            hit_atoms = zip(*atoms)[1]
            image1 = self.generator.image(mol, highlight_atoms=mol_atoms)
            image2 = self.generator.image(hit.molecule, highlight_atoms=hit_atoms)
            return '''
                <tr><th align="center">{mol.identifier}</th><th align="center">{hit.identifier}</th></tr>
                <tr><td>{image1}</td><td>{image2}</td></tr>
                   '''.format(**locals())

        text = [
            '<h2>Similarity to %s with threshold %.2f</h2>\n' % (mol.identifier, self.args.threshold),
            '<table>',
            '\n'.join(one_hit(h) for h in hits if mol.identifier != h.identifier),
            '</table>'
        ]
        self.output.write('\n'.join(text))

################################################################################

if __name__ == '__main__':
    Runner().run()
```

Geometry analysis examples

Generating a HTML report

Note that this script makes use of functionality from the *cookbook utility module*.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

"""
Generate a HTML report along with histograms for any unusual geometric
```

```
features.

For more help use the -h argument.

"""

from __future__ import division, absolute_import, print_function
import argparse
import os

import matplotlib.pyplot as plot

from ccdc.io import MoleculeReader
from ccdc.conformer import GeometryAnalyser
from .utilities import write_html_table

##############################################################################

def write_histogram(file_name, data, ref_val, settings):
    """Write a histogram to a file.

    This function is called by the functions:
      - write_bond_distance_histogram
      - write_valence_angle_histogram
      - write_torsion_angle_histogram
      - write_ring_rmsd_histogram

    """
    # Work out where the histogram bins should be placed on the histogram.
    xs = [settings["low"] + i*settings["bin_width"] for i in range(len(data))]

    # Set up the basic plot.
    fig = plot.figure()
    ax = fig.add_subplot(111)
    bar = ax.bar(xs, data, settings["bin_width"], color='g')
    ax.set_xlim(settings["low"], settings["high"])
    ax.set_xlabel(settings["xlabel"])
    ax.set_ylabel(settings["ylabel"])
    ax.set_title(settings["title"])

    # Add the reference value line.
    ax.add_line(plot.Line2D((ref_val, ref_val), plot.ylim(), color='r'))

    # Add the text describing the query value next to the reference line.
    # Finding a good position for this text requires a little bit of
    # manipulation. If the reference value is to the right of the middle of
    # the plot we place the text on the left hand side of the reference line.
    # Otherwise on the right hand side. We also create some relative offsets
    # so that the text does not cross the frame of the plot or the reference
    # line.
    ref_val_str = 'Value in query: %.2f' % ref_val
    middle = ( settings["low"] + settings["high"] ) / 2.0
    ymin, ymax = plot.ylim()
    xoffset = ( settings["high"] - settings["low"] ) / 100
    yoffset = ( ymin - ymax ) / 80
    ref_y = ymax + yoffset
    if ref_val > middle:
        ref_x = ref_val - xoffset
        ha = 'right'
    else:
```

```
        ref_x = ref_val + xoffset
        ha = 'left'
    ax.text(ref_x, ref_y, ref_val_str, color='r', ha=ha, va='top')
    plot.savefig(file_name)
    plot.close(fig)

def write_bond_distance_histogram(bond, file_name):
    """Write bond distance histogram to a file."""
    settings = {}
    settings["low"] = min(bond.value, bond.minimum) - 0.1
    settings["high"] = max(bond.value, bond.maximum) + 0.1
    diff = settings["high"] - settings["low"]
    settings["bin_width"] = diff/40.0
    settings["title"] = 'Bond lengths for %s' % bond.fragment_label
    settings["xlabel"] = 'Bond length'
    settings["ylabel"] = 'Number of hits'
    data = bond.histogram(settings["bin_width"],
                          settings["low"],
                          settings["high"])
    write_histogram(file_name, data, bond.value, settings)

def write_valence_angle_histogram(angle, file_name):
    """Write bond angle histogram to a file."""
    settings = {}
    settings["low"] = min(angle.value, angle.minimum) - 1
    settings["high"] = max(angle.value, angle.maximum) + 1
    diff = settings["high"] - settings["low"]
    settings["bin_width"] = diff/40.0
    settings["title"] = 'Valence angles for %s' % angle.fragment_label
    settings["xlabel"] = 'Valence angle'
    settings["ylabel"] = 'Number of hits'
    data = angle.histogram(settings["bin_width"],
                           settings["low"],
                           settings["high"])
    write_histogram(file_name, data, angle.value, settings)

def write_torsion_angle_histogram(torsion, file_name):
    """Write torsion angle histogram to a file."""
    settings = {}
    settings["low"] = 0
    settings["high"] = 180
    settings["bin_width"] = 5.0
    settings["title"] = 'Torsion angles for %s' % torsion.fragment_label
    settings["xlabel"] = 'Torsion angle'
    settings["ylabel"] = 'Number of hits'
    data = torsion.histogram(settings["bin_width"],
                             settings["low"],
                             settings["high"])
    write_histogram(file_name, data, abs(torsion.value), settings)

def write_ring_rmsd_histogram(ring, file_name):
    """Write ring RMSD histogram to a file."""
    settings = {}
    settings["low"] = -1.0
    settings["high"] = ring.maximum
    diff = settings["high"] - settings["low"]
    settings["bin_width"] = diff/40.0
    settings["title"] = 'Ring analysis for %s' % ring.fragment_label
    settings["xlabel"] = 'RMSD of torsions from query ring'
    settings["ylabel"] = 'Number of hits'
```

```
    data = ring.histogram(settings["bin_width"],
                          settings["low"],
                          settings["high"])
    write_histogram(file_name, data, 0.0, settings)

################################################################################

HEADERS = (
    'Fragment',
    'Classification',
    'Value',
    'Count',
    'Mean',
    'Median',
    'Z Score',
    'Dmin',
)

TOR_HEADERS = (
    'Fragment',
    'Classification',
    'Value',
    'Count',
    'Mean',
    'Median',
    'Local density',
    'Dmin',
)

RING_HEADERS = (
    'Fragment',
    'Classification',
    'Value',
    'Count',
    'Mean',
    'Median',
    'Dmin',
)

def get_row(frag):
    """Return table data row."""
    if frag.type == 'torsion':
        z_score = frag.local_density
    elif frag.type == 'ring':
        return (
            frag.fragment_label,
            frag.classification,
            '%.3f' % frag.value,
            '%i' % frag.nhits,
            '%.3f' % frag.mean,
            '%.3f' % frag.median,
            '%.3f' % frag.d_min,
        )
    else:
        z_score = frag.z_score
    return (
        frag.fragment_label,
        frag.classification,
        '%.3f' % frag.value,
        '%i' % frag.nhits,
```

```
        '%.3f' % frag.mean,
        '%.3f' % frag.median,
        '%.3f' % z_score,
        '%.3f' % frag.d_min,
    )


################################################################################

def write_histograms(frag_list, out_dir, mol_name, report):
    """Write histogram figures and add image links to report."""
    for frag in frag_list:
        suffix = frag.fragment_label
        file_name = '%s_%s_%s.png' % (mol_name, frag.type, suffix)
        file_path = os.path.join(out_dir, file_name)
        if frag.nhits < 2:
            print(frag.type, suffix, frag.classification, frag.nhits, end=' ')
            print('in distribution, so not writing a histogram')
            continue
        if frag.type == 'bond':
            write_bond_distance_histogram(frag, file_path)
        elif frag.type == 'angle':
            write_valence_angle_histogram(frag, file_path)
        elif frag.type == 'torsion':
            write_torsion_angle_histogram(frag, file_path)
        elif frag.type == 'ring':
            write_ring_rmsd_histogram(frag, file_path)

        report.write('\n'.join([
            '<H3> %s for %s </H3>' % (frag.classification, suffix),
            '<IMG src="%s"></IMG>\n' % file_name
        ]))

def write_report(mol, out_dir):
    """Write histograms and HTML report."""
    identifier = mol.identifier

    report = open(os.path.join(out_dir, identifier + '_report.html'), 'w')
    report.write('\n'.join([
        '<HTML>',
        '<HEAD>',
        '<TITLE>Geometry report for %s</TITLE>' % identifier,
        '</HEAD>',
        '<BODY>',
        '<H1>Geometry report for %s</H1>\n' % identifier
    ]))

    report.write('<H3>Bond distances</H3>\n')
    write_html_table(HEADERS,
                     [get_row(bond) for bond in mol.analysed_bonds],
                     stream=report,
                     border=1)
    write_histograms([b for b in mol.analysed_bonds if b.unusual],
                     out_dir,
                     identifier,
                     report)

    report.write('<H3>Valence angles</H3>\n')
    write_html_table(HEADERS,
                     [get_row(angle) for angle in mol.analysed_angles],
```

```
                    stream=report,
                    border=1)
    write_histograms([a for a in mol.analysed_angles if a.unusual],
                    out_dir,
                    identifier,
                    report)

    report.write('<H3>Torsion angles</H3>\n')
    write_html_table(TOR_HEADERS,
                    [get_row(torsion) for torsion in mol.analysed_torsions],
                    stream=report,
                    border=1)
    write_histograms([t for t in mol.analysed_torsions if t.unusual],
                    out_dir,
                    identifier,
                    report)

#   report.write('<H3>Ring RMSDs</H3>\n')
#   write_html_table(RING_HEADERS,
#                    [get_row(ring) for ring in mol.analysed_rings],
#                    stream=report,
#                    border=1)
#   write_histograms(mol.analysed_rings.unusual(), out_dir, identifier, report)

    report.write('</BODY></HTML>\n')
    report.close()

################################################################################

def main(file_name, out_dir):
    reader = MoleculeReader(file_name)
    engine = GeometryAnalyser()
    engine.settings.summary()
    for mol in reader:
        mol = engine.analyse_molecule(mol)
        write_report(mol, out_dir)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('mol_file', type=str, help="Input molecule file")
    parser.add_argument('out_dir', type=str, help="Output directory")
    args = parser.parse_args()

    if not os.path.isdir(args.out_dir):
        os.mkdir(args.out_dir)
    main(args.mol_file, args.out_dir)
```

Extracting the CSD hits of the analysis distributions

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#
```

```
"""
Analyse the torsion angles of an input molecule and for any unusual torsions
write out the CSD entries that make up the analysis distribution.

This script will take an input molecule in mol2 or sdf file format and perform
a geometry analysis on it. For every unusual torsion it will then write out a
multi mol2 or sdf file containing the CSD entries that make up the analysis
distribution for that particular torsion. Furthermore, for each unusual
torsion a csv file is written to the output directory. This csv file contains
information on the atoms matched in the hit CSD entries.

For more help use the -h argument.

"""

from __future__ import division, absolute_import, print_function
import argparse
import os.path

from ccdc.io import MoleculeReader, MoleculeWriter
from ccdc.conformer import GeometryAnalyser

##############################################################################

CSV_HEADER = '"Refcode","AtId1","AtId2","AtId3","AtId4","Torsion value"\n'

def create_analyser():
    "Return geometry analysis engine with the bond, angle and ring analysis turned off."
    #   Create a geometry analysis instance
    engine = GeometryAnalyser()

    #   Turn off unwanted analyses
    #   Separate instances in settings for each geometry query
    engine.settings.bond.analyse = False
    engine.settings.angle.analyse = False
    engine.settings.ring.analyse = False
    return engine

def main(inputfile, outdir, iformat, oformat):
    "Analyse the molecule and write out the hits for the unusual torsions."

    engine = create_analyser()

    #   Iterate over molecules in the input file.
    mol_reader = MoleculeReader(inputfile, format=iformat)
    for mol in mol_reader:
        checked_mol = engine.analyse_molecule(mol)

        # Iterate over the unusual torsions
        for torsion in [t for t in checked_mol.analysed_torsions if t.unusual]:

            # Write the CSD entries that make up the distribution to a
            # file in the output directory.
            mol_fname = 'torsion_%s_hits.%s' % (torsion.fragment_label, oformat)
            mol_fpath = os.path.join(outdir, mol_fname)
            with MoleculeWriter(mol_fpath) as mol_writer:
                for mol in torsion.hit_molecules:
                    mol_writer.write(mol)
```

```
            # Write out the atom ids of the atoms matched in the hits to a csv
            # file in the output directory.
            csv_fname = 'torsion_%s_hits.csv' % torsion.fragment_label
            csv_fpath = os.path.join(outdir, csv_fname)
            with open(csv_fpath, 'w') as csv_file:
                csv_file.write(CSV_HEADER)
                for hit in torsion.hits:
                    atomId1, atomId2, atomId3, atomId4 = hit.atom_indices
                    line = '"%s",%i,%i,%i,%i,%.2f\n' % (hit.refcode,
                                                        atomId1,
                                                        atomId2,
                                                        atomId3,
                                                        atomId4,
                                                        hit.torsion_angle)
                    csv_file.write(line)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('inputfile', help='The molecule to analyse')
    parser.add_argument('outdir', help='Output directory')
    parser.add_argument('-i', '--input_file_format', choices=['mol2', 'sdf'],
                        default='',
                        help='File format')
    parser.add_argument('-o', '--output_file_format', choices=['mol2', 'sdf'],
                        default='sdf',
                        help='File format')
    args = parser.parse_args()

    # Do some sanity checking of the input arguments.
    if not os.path.isfile(args.inputfile):
        parser.error('%s is not a file.' % args.inputfile)
    if not os.path.isdir(args.outdir):
        os.mkdir(args.outdir)

    # Run the main script.
    main(args.inputfile, args.outdir, args.input_file_format, args.output_file_format)
```

Filtering a set of conformers

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

"""
Filter a set of conformers of a single structure using GeometryAnalyser.

The filtering is based on the idea of minimising the number of unusual
torsions.

The default definition of "unusual" is any torsion angle which has a GeometryAnalyser
local density value of less than 10%. That is less than 10% of the CSD
observed torsions lie within +/- 10 degrees of the input torsion angle. This
cut-off can be adjusted using the "LOCAL_DENSITY" argument.
```

```
The default minimum number of CSD hits for a torsion to be considered at all
is 15. This cut-off can be adjusted using the "MIN_OBS" argument.

For more help use the -h argument.

"""

from __future__ import division, absolute_import, print_function
import sys
import argparse

import ccdc.conformer
import ccdc.io

##############################################################################

def create_analyser():
    """Create a GeometryAnalyser engine to analyse the conformers."""
    engine = ccdc.conformer.GeometryAnalyser()

    # We are only interested doing a analysis of the torsions.
    engine.settings.bond.analyse = False
    engine.settings.angle.analyse = False
    engine.settings.ring.analyse = False

    # Also we do not want to use generalisation.
    engine.settings.generalisation = False
    # Only the organic subset, i.e. exclude organometallic
    engine.settings.organometallic_filter = 'Organic'
    return engine

def main(infile, outfile, local_density_cutoff, min_obs):
    """The main program for filtering a set of conformers."""
    engine = create_analyser()

    # Set the current best (minimum number of unusual torsions) to a large
    # number.
    best_so_far = sys.maxsize

    # Open a file to write molecules to.
    out = ccdc.io.MoleculeWriter(outfile)
    with ccdc.io.MoleculeReader(infile) as mol_reader:
        for mol in mol_reader:

            # Do the analysis.
            checked_mol = engine.analyse_molecule(mol)

            # Find the number of unusual torsions according to the input
            # criteria.
            unusual_torsions = [
                t for t in checked_mol.analysed_torsions
                  if (t.local_density < local_density_cutoff
                      and t.nhits > min_obs)
            ]
            num_unusual_torsions = len(unusual_torsions)

            # Should the conformer be kept or discarded.
            if num_unusual_torsions > best_so_far:
                continue
```

```
            elif num_unusual_torsions == best_so_far:
                out.write(checked_mol)
            elif num_unusual_torsions < best_so_far:
                # We have found something better, which means we have to
                # discard all the conformers written out already. We therefore
                # close the output writer and open it again.
                out.close()
                out = ccdc.io.MoleculeWriter(outfile)
                out.write(checked_mol)

                # Make sure that we update the variable that we are using for
                # the comparison.
                best_so_far = num_unusual_torsions

if __name__ == '__main__':
    # Set up the command line parser
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('infile', type=str, help="Input molecule file")
    parser.add_argument('outfile', type=str, help="Output molecule file")
    parser.add_argument('-l', '--local_density', type=float, default=10.0,
        help="Local density cutoff for defining an unusual torsion")
    parser.add_argument('-m', '--min_obs', type=int, default=15,
        help="Minimum number of observations for a torsion to be considered")

    # Parse the command line arguments
    args = parser.parse_args()

    # Run the conformer filtering
    main(args.infile, args.outfile, args.local_density, args.min_obs)
```

Powder pattern examples

> ### *Note*
>
> The powder packing features are available only to CSD-Materials and CSD-Enterprise users.

Comparing powder patterns

Note that this script makes use of functionality from the *cookbook utility module*.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
Generate a HTML report with powder pattern comparisons.

Compare a powder pattern from a .xye file with simulations derived from
crystals in another file.
```

```
The resulting HTML report and the powder pattern images are written out to the
specified output directory.
'''
################################################################################

import sys
import os
import argparse
import math

from ccdc.io import CrystalReader
from ccdc.descriptors import PowderPattern
from .utilities import Lineplot


################################################################################

class Runner(argparse.ArgumentParser):
    '''Class for producing the powder similarity HTML report.'''

    def __init__(self):
        '''Define the command line interface.'''

        super(Runner, self).__init__(description=__doc__)
        self.add_argument(
            'xye_file', type=str,
            help='xye format experimental powder'
        )
        self.add_argument(
            'crystal_file', type=str,
            help='file of crystal structures to compare'
        )
        self.add_argument(
            'output_directory', type=str,
            help='output directory for images and html index'
        )
        self.add_argument(
            '-s', '--start', default=0, type=int,
            help='Crystal at which to start'
        )
        self.add_argument(
            '-e', '--end', default=sys.maxint, type=int,
            help='Crystal at which to end'
        )
        self.args = self.parse_args()
        if not os.path.exists(self.args.xye_file):
            raise argparse.ArgumentError(
                'xye file does not exist: %s' % self.args.xye_file)
        if not os.path.exists(self.args.crystal_file):
            raise argparse.ArgumentError(
                'crystal file does not exist: %s' % self.args.crystal_file)
        if not os.path.exists(self.args.output_directory):
            os.makedirs(self.args.output_directory)

    def make_title(self):
        '''Return the report title.'''
        return 'Comparison of %s with %s' % (os.path.basename(self.args.xye_file),
                                             os.path.basename(self.args.crystal_file))
```

```
    def make_plot(self, patt, name, ref=None, file_name=None, thumb=False):
        '''Generate a plot from the powder pattern and return the file name.'''
        import matplotlib.pyplot as plot
        if file_name is None:
            file_name = str(os.path.join(self.args.output_directory,
                                         '%s.png' % name))
```

```
        else:
            file_name = str(os.path.join(self.args.output_directory,
                                         '%s.png' % file_name))

        # Crete the matplotlib plot.
        pl = Lineplot(
            file_name=file_name,
            xlabel=r'2$\theta$',
            ylabel='Intensity',
            title='PXRD %s' % name
        )

        # Add the first powder pattern.
        pl.add_plot(patt.two_theta, patt.intensity, label='Intensity')

        # Add the second powder pattern if provided.
        if ref is not None:
            pl.add_plot(ref.two_theta, ref.intensity, label='Reference', color='red')

        # If this is a thumbnail, configure, write and return file name.
        if thumb:
            pl.title = pl.xlabel = pl.ylabel = ''
            plot.tick_params(
                axis='x', which='both', bottom='off', top='off', labelbottom='off'
            )
            plot.tick_params(
                axis='y', which='both', left='off', right='off', labelleft='off'
            )
            pl.fig.set_size_inches(3., 1.)
            pl.write()
            return os.path.basename(file_name)

        # At this point it is not a thumbnail, so add axis, title, etc.
        artists = [
            plot.Line2D((0,0),(0,1), color='b'),
        ]
        labels = ['Intensity']
        if patt.tick_marks is not None:
            absent = [tm.two_theta for tm in patt.tick_marks
                                   if tm.is_systematically_absent]
            present = [tm.two_theta for tm in patt.tick_marks
                                    if not tm.is_systematically_absent]
            if absent:
                pl.add_plot(absent, [-250.0] * len(absent), linestyle=' ',
                            marker='|', markersize=10,
                            label='Systematic absences')
                artists.append(
                    plot.Line2D((0,0),(0,1), color='g')
                )
                labels.append('Systematic absences')
            if present:
                pl.add_plot(present, [-250.0]* len(present), linestyle=' ',
                            marker='|', markersize=10, color='pink',
                            label='Allowed reflections')
                artists.append(
                    plot.Line2D((0,0),(1,0), color='pink')
                )
                labels.append('Allowed reflections')
            if absent or present:
                pl.axes.get_yaxis().set_view_interval(-500, 10000, True)
```

```
            pl.axes.get_yaxis().set_ticks([1000*i for i in range(11)])
            pl.axes.legend(artists, labels)
        pl.write()
        return os.path.basename(file_name)

    def make_table(self):
        '''Returns a string containing the HTML table for the report.'''

        def do_one(c):
            '''Return a tuple containing the identifier and the powder pattern.'''
            try:
                p = PowderPattern.from_crystal(c)
            except RuntimeError:
                p = None
            return c.identifier, p

        l = [
            do_one(self.reader[i]) for i in
            range(self.args.start, min(len(self.reader), self.args.end))
        ]

        # Store the: similarity, identifier and powder pattern
        self.powder_sims = [
            (self.powder.similarity(p), i, p) for i, p in l if p is not None
        ]
        self.powder_sims = [
            (s, i, p) for s, i, p in self.powder_sims if not math.isnan(s)
        ]
        self.powder_sims.sort(reverse=True)

        def row(s, i, p):
            '''Return a HTML table row.

            :param s: similarity
            :param i: identifier
            :param p: pattern
            '''
            refcode = '<TD>%s</TD>' % i
            similarity = '<TD align="right">%.4f</TD>' % s
            thumbnail_img = '<IMG src="%s_thumbnail.png"/>' % i
            thumbnail = '<TD><A HREF="#%s_image">%s</A></TD>' % (i, thumbnail_img)
            overlay_img = '<IMG src="%s_overlay_thumbnail.png">' % i
            overlaid = '<TD><A HREF="#%s_overlay_image">%s</A></TD>' % (i, overlay_img)
            return '<TR>%s%s%s%s</TR>' % (refcode, similarity, thumbnail, overlaid)

        ret = [
            '<TABLE border=1>',
            '<TR><TH>Refcode</TH><TH>Similarity</TH><TH>Thumbnail</TH><TH>Overlaid</TH></TR>',
            '\n'.join(row(*x) for x in self.powder_sims),
            '</TABLE>'
        ]
        for s, i, p in self.powder_sims:
            self.make_plot(p, i, file_name='%s_thumbnail' % i, thumb=True)
        for s, i, p in self.powder_sims:
            self.make_plot(p, i, ref=self.powder,
                           file_name='%s_overlay_thumbnail' % i, thumb=True)
        return '\n'.join(ret)

    def make_plots(self):
        '''Create the plots for the HTML report.
```

```
        Returns a HTML string containing references to the plots.
        '''
        def make_one(s, i, p):
            '''Make one pair of plots.
```

```
            :param s: similarity
            :param i: identifier
            :param p: pattern
            :returns: HTML string containing containing references to the plots
            '''
            l = [
                '<DIV>',
                '<IMG id="%s_image" src="%s.png"/>' % (i, i),
                '<IMG style="float:right" id="%s_overlay_image" src="%s_overlay.png"/>' % (i, i),
                '</DIV>',
            ]
            self.make_plot(p, i)
            self.make_plot(p, i, self.powder, file_name='%s_overlay' % i)
            return '\n'.join(l)
        l = [
            make_one(*x) for x in self.powder_sims
        ]
        return '\n'.join(l)

    def run(self):
        '''Generate the HTML report.'''

        # Define the reference powder pattern and the crystals to be compared.
        self.powder = PowderPattern.from_xye_file(self.args.xye_file)
        self.reader = CrystalReader(self.args.crystal_file)

        # Create a plot of the reference powder pattern.
        ref_pattern_plot_file = self.make_plot(self.powder,
            'Reference %s' % os.path.basename(self.args.xye_file),
                                               file_name='reference')

        # Create a list containing the content of the HTML report.
        l = [
            '<HTML><HEAD><TITLE>%s</TITLE></HEAD>' % self.make_title(),
            '<BODY>',
            '<H1>%s</H1>' % self.make_title(),
            '<IMG src="%s"/>' % ref_pattern_plot_file,
            '<HR>',
            self.make_table(),
            '<HR>',
            self.make_plots(),
            '</BODY></HTML>\n'
        ]

        # Write out the content of the list to the output index.html file.
        with open(os.path.join(self.args.output_directory, 'index.html'), 'w') as w:
            w.write('\n'.join(l))


################################################################################

if __name__ == '__main__':
    Runner().run()
```

Calculating many powder patterns

Note that this script makes use of functionality from the *cookbook utility module*.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
```

```
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
Create and saves powder patterns from all crystals in a file.

Produces a HTML report in the output directory alongside the powder pattern
files.
'''

from __future__ import division, absolute_import, print_function
import sys
import os
import argparse

from ccdc.io import CrystalReader
from ccdc.descriptors import PowderPattern
from ccdc.diagram import DiagramGenerator
from .utilities import Lineplot

################################################################################

class Runner(argparse.ArgumentParser):
    '''Runs the job.'''

    def __init__(self):
        '''Define the command line interface.'''
        super(Runner, self).__init__(description=__doc__)

        self.add_argument(
            'input_file', type=self.valid_file,
            help='input file containing crystals'
        )
        self.add_argument(
            'output_directory', type=self.valid_directory,
            help='output directory to place files'
        )
        self.add_argument(
            '-d', '--diagram', action='store_true', default=False,
            help='Whether to generate diagrams'''
        )
        self.add_argument(
            '-x', '--xye-file', action='store_false', default=True,
            help='Cancel the writing of xye files'
        )
        self.add_argument(
            '-r', '--raw-file', action='store_false', default=True,
            help='Cancel the writing of raw files'
        )
        self.add_argument(
            '-t', '--table', action='store_true', default=False,
            help='Write a cross-similarity table'
        )
        self.add_argument(
            '-c', '--count', default=sys.maxsize, type=int,
            help='Count of structures to process'
        )
        self.args = self.parse_args()
        l = [
            '<HTML><HEAD><TITLE>PXRD for %s</TITLE></HEAD>' % os.path.basename(self.args.input_file),
            '<BODY>'
        ]
        self.diag = DiagramGenerator()
        self.all_patts = dict()
        for i, c in enumerate(CrystalReader(self.args.input_file)):
            if i >= self.args.count:
                break
            try:
                patt = PowderPattern.from_crystal(c)
                self.all_patts[c.identifier] = patt
```

```
            except RuntimeError as e:
                patt = None

            if patt is not None and self.args.xye_file:
                patt.write_xye_file(os.path.join(self.args.output_directory, '%s.xye' % c.identifier))
            if patt is not None and self.args.raw_file:
                patt.write_raw_file(os.path.join(self.args.output_directory, '%s.raw' % c.identifier))

            l.append('<DIV>')
            if self.args.diagram:
                l.append(
                    '<IMG src="%s"></IMG>' % self.make_diagram(c)
                )
            if patt is not None:
                l.append('<A NAME="%s">' % c.identifier)
                l.append(
                    '<IMG src="%s"></IMG>' % self.make_plot(patt, c.identifier)
                )
            else:
                l.append('<P>No powder simulation for %s: %s</P>' % (c.identifier, e))
            l.append('</DIV>')
        if self.args.table:
            l.append(self.make_table())
        l.append('</BODY></HTML>')
        f = open(os.path.join(self.args.output_directory, 'index.html'), 'w')
        f.write('\n'.join(l))
        f.close()

    def make_table(self):
        '''Generates a comparison table of all powder simulations.'''
        names = sorted(self.all_patts.keys())
        def do_row(k):
            x = [
                '<TR><TH>',
                '<A HREF="#%s">%s</A>' % (k, k),
                '</TH><TD>',
                '</TD><TD>'.join('%.3f' % self.all_patts[k].similarity(self.all_patts[p]) for p in names),
                '</TD></TR>'
            ]
            return ''.join(x)

        l = [
            '<TABLE border=1>',
            '<TR><TH> </TH><TH>',
            '</TH><TH>'.join('<A HREF="#%s">%s</A>' % (k, k) for k in names),
            '</TH></TR>',
            '\n'.join(do_row(k) for k in names),
            '</TABLE>'
        ]
        return '\n'.join(l)

    def make_diagram(self, crystal):
        '''Generates a diagram from the crystal.'''
        img = self.diag.image(crystal)
        fname = os.path.join(self.args.output_directory, '%s_diagram.png' % crystal.identifier)
        if img:
            img.save(fname)
        return os.path.basename(fname)

    def make_plot(self, patt, name):
```

```
        '''Generate a plot from the powder pattern.'''
        import matplotlib.pyplot as plot
        file_name = str(os.path.join(self.args.output_directory, '%s.png' % name))
        pl = Lineplot(
            file_name=file_name,
            xlabel=r'2$\theta$',
            ylabel='Intensity',
            title='PXRD %s' % name
        )
        pl.add_plot(patt.two_theta, patt.intensity, label='Intensity')
```

```
        artists = [
            plot.Line2D((0, 0), (0, 1), color='b'),
        ]
        labels = ['Intensity']
        absent = [tm.two_theta for tm in patt.tick_marks if tm.is_systematically_absent]
        present = [tm.two_theta for tm in patt.tick_marks if not tm.is_systematically_absent]
        if absent:
            pl.add_plot(absent, [-250.0] * len(absent), linestyle=' ',
                        marker='|', markersize=10,
                        label='Systematic absences')
            artists.append(
                plot.Line2D((0, 0), (0, 1), color='g')
            )
            labels.append('Systematic absences')
        if present:
            pl.add_plot(present, [-250.0]* len(present), linestyle=' ',
                        marker='|', markersize=10, color='pink',
                        label='Allowed reflections')
            artists.append(
                plot.Line2D((0, 0), (1, 0), color='pink')
            )
            labels.append('Allowed reflections')
        if absent or present:
            pl.axes.get_yaxis().set_view_interval(-500, 10000, True)
            pl.axes.get_yaxis().set_ticks([1000*i for i in range(11)])
            pl.axes.legend(artists, labels)
        pl.write()
        return os.path.basename(file_name)

    def valid_file(self, f):
        '''Return existence of a file.'''
        if os.path.exists(f):
            return f
        self.error('File %s does not exist' % f)

    def valid_directory(self, f):
        '''Return existence of a directory, after trying to make it if necessary.'''
        if os.path.exists(f) and os.path.isdir(f):
            return f
        if not os.path.exists(f):
            os.makedirs(f)
            return f
        self.error('Directory %s does not exist and could not be created' % f)

##############################################################################

if __name__ == '__main__':
    Runner()
```

Packing similarity examples

> ### *Note*
>
> The crystal packing similarity feature is available only to CSD-Discovery, CSD-Materials and CSD-Enterprise users.

Comparing similarity measures

This example searches the CSD for structures similar to a given entry, then calculates the molecular-, crystal- and powder pattern simulation similarities, presenting them in an HTML table with a diagram of the structure. For example **`this`** which was created from the command line 'python ccdc/examples/packing_similarity.py ZERLUD --threshold 0.8'.

```
#!/usr/bin/env python
#
```

```
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
    packing_similarity.py   - find similar molecules and report on crystal packing and powder simulation similarities.

    The CSD will be searched for any similar molecules to the given refcode(s).  Their Crystal packing
    similarity and Powder Simulation similarity will be reported in an HTML table.
'''
#------------------------------------------------------------------------------

import argparse

from ccdc.io import EntryReader
from ccdc.search import SimilaritySearch
from ccdc.crystal import PackingSimilarity
from ccdc.descriptors import PowderPattern
from ccdc.diagram import DiagramGenerator

#------------------------------------------------------------------------------

class Runner(argparse.ArgumentParser):
    '''Interprets arguments, and runs the script.'''
    def __init__(self):
        super(self.__class__, self).__init__(description=__doc__)
        self.add_argument(
            'refcodes', nargs='+', help='Refcode(s) for which to perform the similarity search and comparison'
        )
        self.add_argument(
            '-t', '--threshold', default=0.9, type=float, help='Similarity search threshold to use.'
        )
        self.parse_args(namespace=self)

    def run(self):
        '''Run all the comparisons.'''
        self.csd = EntryReader('csd')
        self.packing_sim = PackingSimilarity()
        self.packing_sim.settings.allow_molecular_differences = True
        self.diagram_generator = DiagramGenerator()
        self.diagram_generator.settings.return_type = 'SVG'
        print '<H2>Similarity measures</H2>'
        print '<TABLE border="1">'
        print '<TR><TH>Refcode</TH><TH>Diagram</TH><TH>Molecular</TH><TH>Packing</TH><TH>Powder Sim.</TH></TR>'
        for refcode in self.refcodes:
            self.run_one(refcode)
        print '</TABLE>'

    def run_one(self, refcode):
        '''Single comparison.'''
        crystal = self.csd.crystal(refcode)
        powder_sim = PowderPattern.from_crystal(crystal)
        mol = crystal.molecule
        diagram = self.diagram_generator.image(self.csd.entry(refcode))
        search = SimilaritySearch(mol, self.threshold)
```

```
        hits = search.search()
        print '<TR><TH>%s</TH><TD>%s<TD colspan=3>%d hits</TD></TR>' % (
            refcode, diagram, len(hits)
        )
        #print '%8s: %d hits' % (refcode, len(hits))
        for h in hits:
            packing_sim = self.packing_sim.compare(crystal, h.crystal)
            other_powder = PowderPattern.from_crystal(h.crystal)
            powder = powder_sim.similarity(other_powder)
            other_diagram = self.diagram_generator.image(h.entry)
            if packing_sim is None:
                pack = '---'
                #print '\t%-8s %.2f None' % (h.identifier, h.similarity)
            else:
                pack = '%.2f (%d)' % (packing_sim.rmsd, packing_sim.nmatched_molecules)
```

```
                #print '\t%-8s %.2f %.2f (%d)' % (
                #    h.identifier, h.similarity, packing_sim.rmsd, packing_sim.nmatched_molecules
                #)
```

```
            print '<TR><TD>%s</TD><TD>%s<TD>%.2f</TD><TD>%s</TD><TD>%.2f</TD></TR>' % (
                h.identifier, other_diagram, h.similarity, pack, powder
            )

######################################################################

if __name__ == '__main__':
    Runner().run()
```

Utility examples

Matplotlib plots and HTML generators

```
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

"""
Utilities used in the other user stories.
"""

################################################################################

from __future__ import division, absolute_import, print_function
import sys
from contextlib import contextmanager

import matplotlib.pyplot as plot
import numpy

################################################################################

class Plot(object):
    '''thin wrapper around a matplotlib plot'''
    def __init__(self, title='', xlabel='', ylabel='', file_name='scatter.png'):
        '''initialise the plot'''
        self.fig = plot.figure()
        self.axes = self.fig.add_subplot(1, 1, 1)
        self.title = title
        self.xlabel = xlabel
        self.ylabel = ylabel
        self.file_name = file_name

    def __del__(self):
        '''Ensure memory is freed.'''
        plot.close(self.fig)

    def _axes_property(name, help):
        '''define a property'''
        return property(
            lambda self: getattr(self.axes, 'get_' + name)(),
            lambda self, val: getattr(self.axes, 'set_' + name)(val),
            help
        )
```

```
    title = _axes_property('title', 'The title of the plot')
    xlabel = _axes_property('xlabel', 'The label for the X axis')
    ylabel = _axes_property('ylabel', 'The label for the Y axis')

    def write(self):
        self.fig.savefig(self.file_name)

class Scatterplot(Plot):
    '''A simple scatter plot'''
    def add_plot(self, xs, ys, color='green'):
        '''Add some scattered points'''
        self.axes.scatter(xs, ys, color=color)

    def annotate(self, x, y, text, color='red'):
        '''Mark a point'''
        self.add_plot([x], [y], color=color)
        xmin, xmax = self.axes.get_xlim()
        ymin, ymax = self.axes.get_ylim()
        xmid = (xmin + xmax)/2.
        ymid = (ymin + ymax)/2.
        xoff = (xmax - xmin)/10.
        yoff = 0
        ha = 'left'
        va = 'center'
        if x > xmid:
            xoff *= -1
            ha = 'right'
        self.axes.annotate(
            text, (x, y), (x+xoff, y+yoff),
            ha=ha, va=va,
            arrowprops=dict(
                arrowstyle='simple',
                facecolor=color
            )
        )

class Histogram(Plot):
    '''A simple histogram'''
    def add_plot(self, data, color='green', bins=40):
        '''Histogram the data and add to the plot'''
        ys, xs = numpy.histogram(data, bins=bins)
        xmin = min(data)
        xmax = max(data)
        width = (xmax - xmin)/bins
        self.axes.bar(xs[:-1], ys, width, color=color)

    def add_histogram(self, data, low, high, color='green'):
        '''When the data has been histogrammed already'''
        width = (high - low)/float(len(data))
        xs = [low + i*width for i in range(len(data))]
        self.axes.set_xlim(low, high)
        self.axes.bar(xs, data, width, color=color)

    def annotate(self, x, text, color='red', yoff=0.8):
        self.axes.add_line(plot.Line2D((x, x), self.axes.get_ylim(), color=color))
        xmin, xmax = self.axes.get_xlim()
        xmid = (xmin + xmax)/2.
        xoff = (xmax - xmin)/10.
        ha = 'left'
```

```
        if x > xmid:
            xoff *= -1
            ha = 'right'
        yoff *= sum(self.axes.get_ylim())
        self.axes.annotate(
            text, (x, yoff), (x + xoff, yoff), color=color,
            ha=ha, va='center',
            arrowprops=dict(
                arrowstyle='simple',
                facecolor=color
            )
        )

class PolarScatterplot(Plot):
    '''Simple polar scatter'''
    def __init__(self, **kw):
        '''Initialise'''
        Plot.__init__(self, **kw)
        self.fig.delaxes(self.axes)
        self.axes = self.fig.add_subplot(1, 1, 1, polar=True)

    def add_plot(self, theta, r, color='green'):
        '''Add a plot with theta in degrees'''
        self.axes.scatter((math.radians(x) for x in theta), r, c=color)

class Lineplot(Plot):
    '''Simple line plot.'''
    def __init__(self, **kw):
        Plot.__init__(self, **kw)
        t, x, y = self.title, self.xlabel, self.ylabel
        self.fig.delaxes(self.axes)
        self.axes = self.fig.add_subplot(1, 1, 1)
        self.title, self.xlabel, self.ylabel = t, x, y

    def add_plot(self, xs, ys, **kw):
        '''Add a plot.

        Keywords may be any arguments accepted by :class:`matplotlib.Line'''
        self.axes.plot(xs, ys, **kw)

################################################################################


################################################################################

class Curry(object):
    """Bind a callable to arguments and keywords."""

    def __init__(self, f, *args, **kw):
        """Store callable, args and keywords."""
        self.f = f
        self.args = args
        self.kw = kw

    def __call__(self, *args, **kw):
        """
        Extend the bound arguments and keywords, then call and return result.
        """
        d = dict()
        d.update(self.kw)
```

```
        d.update(kw)
        return self.f(*(self.args + args), **d)

@contextmanager
def element(tag, stream=sys.stdout, **attribs):
    """Write HTML elements with attributes to the given stream."""
    stream.write(
        '<%s %s>' % (tag, ' '.join('%s="%s"' % (k, v)
        for k, v in attribs.items())
    ))
    yield stream
    stream.write('</%s>' % tag)

html_table  = Curry(element, 'table')
html_row    = Curry(element, 'tr')
html_header = Curry(element, 'th')
html_datum  = Curry(element, 'td')

def write_html_table(headers, data, stream=sys.stdout, **attribs):
    """Write HTML table with attributes to the given stream."""
    with html_table(stream=stream, **attribs) as s:
        with html_row(stream=s):
            for h in headers:
                with html_header(stream=s):
                    s.write(h)
        for row in data:
            with html_row(stream=s):
                for d in row:
                    with html_datum(stream=s):
                        s.write(str(d))
```

Docking examples

Simple docking

This example shows how a docking may be performed from a protein and a set of ligands. It is not intended to be a production-quality protein-ligand docking, but to exemplify the technology.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#
##############################################################################
'''
    simple_docking.py   -   reads a protein, performs a minimal cleaning up, defines a docking
        and runs the docker.

    Note that any realistic docking would require a great deal of thought to prepare the protein
    properly.  This example shows the minimum that needs to be done to define a docking.
'''
##############################################################################

import os
```

```
import argparse

from ccdc.io import EntryWriter, MoleculeReader
from ccdc.protein import Protein
from ccdc.docking import Docker
from ccdc.utilities import _find_test_file, _test_output_dir

##############################################################################
```

```
class Runner(argparse.ArgumentParser):
    '''Parses arguments, runs the docking.'''
    def __init__(self):
        '''Defines the arguments.'''
        super(self.__class__, self).__init__(description=__doc__)
        self.add_argument(
            'protein_file_name',
            help='The protein file name'
        )
        self.add_argument(
            '-l', '--ligand-file-name',
            help='Ligand file. If not supplied the substrate ligand will be docked.'
        )
        self.add_argument(
            '-s', '--speed', default='very fast', choices=('very fast', 'fast', 'medium', 'slow', 'very slow'),
            help='How thorough the docking attempts should be.  The values correspond to 10%%, 25%%, 50%%, 75%% and 100%% autoscaling of docking parameters.'
        )
        self.add_argument(
            '-n', '--ndocks', default=10, type=int,
            help='Number of docking attempts to run.'
        )
        self.add_argument(
            '-d', '--directory',
            help='Output directory for the docking. Default is a temporary directory.'
        )
        self.add_argument(
            '-f', '--fitness-function', choices=('plp', 'asp', 'chemscore', 'goldscore'), default='plp',
            help='Fitness function to use (default plp).'
        )
        self.add_argument(
            '-r', '--rescore-function', choices=('plp', 'asp', 'chemscore', 'goldscore'),
            help='Rescore function to use (default None)'
        )
        self.add_argument(
            '-b', '--binding-site',
            help='Binding site definition for docking (default a sphere of radius 12A from the origin: (0,0,0,12))'
        )

        self.args = self.parse_args()

    def run(self):
        '''Run the docking.'''
        self.create_docker()
        self.prepare_protein()
        self.prepare_ligands()
        self.prepare_binding_site()
        self.dock()

    def create_docker(self):
        '''Create a docker.'''
        self.docker = Docker()
        self.settings = self.docker.settings
        if self.args.directory:
            if not os.path.isdir(self.args.directory):
                os.makedirs(self.args.directory)
        else:
            self.args.directory = _test_output_dir(remove=False)
        self.settings.output_directory = self.args.directory
        self.settings.output_file = 'docked_ligands.mol2'
```

```
        self.settings.output_format = 'mol2'
        self.settings.autoscale = {
            'very fast': 10,
            'fast': 25,
            'medium': 50,
            'slow': 75,
            'very slow': 100
        }[self.args.speed]
        self.settings.fitness_function = self.args.fitness_function
        if self.args.rescore_function:
            self.settings.rescore_function = self.args.rescore_function

    def prepare_binding_site(self):
        '''Define the binding site.'''
        if not self.args.binding_site:
            if self.ligand is None:
                self.settings.binding_site = self.settings.BindingSite(
                        (0.0,0.0,0.0), 12.0
                )
            else:
                self.settings.binding_site = self.settings.BindingSite(
                   self.ligand.centre_of_geometry(), 12.0
                )
        else:
            (x,y,z) = [float(i) for i in self.args.binding_site.rstrip().split(',')[0:3]]
            radius = float(self.args.binding_site.rstrip().split(',')[-1])
```

```
            self.settings.binding_site = self.settings.BindingSite(
                (x,y,z), radius
            )

    def prepare_protein(self):
        '''Prepare the protein.'''
        self.protein = Protein.from_file_name(self.args.protein_file_name)
        self.protein.remove_all_waters()
        ligands = self.protein.ligands
        for l in ligands:
            self.protein.remove_ligand(l.identifier)
```

```
        if len(ligands):
            self.ligand = ligands[0]
        else:
            self.ligand = None
        self.protein.remove_unknown_atoms()
        self.protein.add_hydrogens()
        protein_file_name = os.path.join(
            self.settings.output_directory, 'clean_%s.mol2' % self.protein.identifier
        )
        with EntryWriter(protein_file_name) as writer:
            writer.write(self.protein)
        self.settings.add_protein_file(protein_file_name)

    def prepare_ligands(self):
        '''Write an appropriate ligand file to the docking directory, if required.'''
        if not self.args.ligand_file_name:
            if self.ligand is None:
                raise RuntimeError('No ligand file given and no ligands in the protein.')
            self.args.ligand_file_name = os.path.join(
                self.settings.output_directory, '%s.mol2' % self.ligand.identifier.split(':')[-1]
            )
            with EntryWriter(self.args.ligand_file_name) as writer:
                writer.write(self.ligand)
        else:
            self.ligand = MoleculeReader(self.args.ligand_file_name)[0]
        self.settings.binding_site = self.settings.BindingSite(
            self.ligand.centre_of_geometry(), 12.0
        )
        self.settings.add_ligand_file(self.args.ligand_file_name, self.args.ndocks)

    def dock(self):
        '''Do the docking.'''
        settings_file = os.path.join(
            self.settings.output_directory, 'settings.conf'
        )
        self.settings.write(settings_file)
        results = self.docker.dock(settings_file)
        print 'Docking completed.  Results are in %s' % self.settings.output_directory

##############################################################################

if __name__ == '__main__':
    runner = Runner()
    runner.run()
```

Interactive docking

This example shows how the results of a docking may be used to inform further dockings in an iterative fashion.

```
#!/usr/bin/env python
#
# This script can be used for any purpose without limitation subject to the
# conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
#
# This permission notice and the following statement of attribution must be
# included in all copies or substantial portions of this script.
#
# 2015-06-17: created by the Cambridge Crystallographic Data Centre
#

'''
    similarity_docking.py - search the CSD for similar structures to a known
    ligand, dock them, saving structures which score better than the known ligand.

    This exemplifies using an interactive docker.
'''
##############################################################################

import sys
import os
import argparse

from ccdc.io import EntryReader, EntryWriter
from ccdc.search import SimilaritySearch
from ccdc.docking import Docker
from ccdc.entry import Entry

##############################################################################################

class Runner(argparse.ArgumentParser):
```

```
    '''Defines arguments and runs the job.'''
    def __init__(self):
        '''Defines the arguments.'''
        super(self.__class__, self).__init__(description=__doc__)
        self.add_argument(
            'conf_file_name',
            help='The configuration file to use.'
        )
        self.add_argument(
            '-l', '--ligand-file-name', default=None,
            help='Ligand to use in the docking'
        )
        # Docking parameters
```

```
        self.add_argument(
            '-f', '--fitness-function', default=None,
            help='The fitness function to use.'
        )
        self.add_argument(
            '-n', '--ndocks', default=0, type=int,
            help='The number of docking attempts to make.'
        )
        self.add_argument(
            '-o', '--output-directory',
            help='Directory in which results will be stored.'
        )
        self.add_argument(
            '-s', '--speed', default=None, choices=('very fast', 'fast', 'medium', 'slow', 'very slow'),
            help='How thorough the docking attempts should be.  The values correspond to 10%%, 25%%, 50%%, 75%% and 100%% autoscaling of docking parameters.'
        )

    def get_arguments(self):
        '''Extract and process the arguments.'''
        self.args = self.parse_args()
        self.settings = Docker.Settings.from_file(self.args.conf_file_name)
        if self.args.output_directory:
            self.settings.output_directory = self.args.output_directory
        if self.args.fitness_function is not None:
            self.settings.fitness_function = self.args.fitness_function
        if self.args.speed is not None:
            self.settings.autoscale = {
                'very fast': 10,
                'fast': 25,
                'medium': 50,
                'slow': 75,
                'very slow': 100
            }[self.args.speed]
        self.ligand_info = self.settings.ligand_files
        if self.args.ligand_file_name is None:
            self.args.ligand_file_name = self.ligand_info[0].file_name
        self.settings.clear_ligand_files()
        self.settings.set_hostname(ndocks=self.args.ndocks)
        self.docker = Docker(settings=self.settings)
        if not os.path.isdir(self.settings.output_directory):
            os.makedirs(self.settings.output_directory)
        self.winner_writer = EntryWriter(
            os.path.join(self.settings.output_directory, 'winners.mol2')
        )
        self.better_writer = EntryWriter(
            os.path.join(self.settings.output_directory, 'better.mol2')
        )

    def run(self):
        '''Runs the job.'''
        self.get_arguments()
        self.session = self.docker.dock(
            os.path.join(self.settings.output_directory, 'settings.conf'),
            'interactive'
        )
        self.run_substrate()
        self.done = set()
        dock_index = 1
        while True:
            hits = self.search()
```

```
            if not hits:
                break
            for h in hits:
                self.done.add(h.identifier)
                mol = h.molecule
                if mol.all_atoms_have_sites:
                    mol.assign_bond_types()
                    mol.add_hydrogens()
                    entry = Entry.from_molecule(mol)
                    poses = self.session.dock(entry)
                    dock_index += 1
                    if poses:
                        score, pose = self.best(poses)
                        for inx, p in enumerate(poses):
                            with EntryWriter(
                                os.path.join(
                                    self.settings.output_directory,
                                    'gold_soln_CSD_API_m%d_%d.mol2' % (dock_index, inx+1)
                                )
                            ) as w:
                                w.write(pose)
                        if score > self.best_score:
                            print '%-8s: %.3f Winner!' % (h.identifier, score)
                            self.best_score, self.best_pose = score, pose
                            self.winner_writer.write(self.best_pose)
                        else:
```

```
                            print '%-8s: %.3f' % (h.identifier, score)
                            if score > self.substrate_score:
                                self.better_writer.write(pose)
                    else:
                        print '%-8s: Failed to dock' % h.identifier
                else:
                    print '%-8s: No 3d structure' % h.identifier

    def search(self):
        '''Similarity search the csd.'''
        threshold = 1.0
```

```
        hits = []
        mol = self.best_pose.molecule
        mol.standardise_aromatic_bonds()
        while not hits and threshold > 0.5:
            threshold -= 0.1
            simsearcher = SimilaritySearch(mol, threshold)
            simsearcher.settings.only_organic = True
            simsearcher.settings.has_3d_coordinates = True
            hits = [h for h in simsearcher.search() if h.identifier not in self.done]

            if hits:
                def match_components(h):
                    if len(h.molecule.components) > 1:
                        for m in h.molecule.components:
                            m.identifier = h.identifier
                            hh = simsearcher.search(m)
                            if hh:
                                return hh[0]
                        else:
                            return None
                            #raise RuntimeError('Similarity search failed to match any component')
                    else:
                        return h

                l = [x for x in [match_components(h) for h in hits] if x is not None]
                print '%d new hits with threshold %.2f against %s' % (len(l), threshold, mol.identifier)
                return l

    def run_substrate(self):
        '''Runs the substrate, for the comparison.'''
        self.substrate = EntryReader(self.args.ligand_file_name)[0]
        poses = self.session.dock(self.substrate)

        if not poses:
            print 'FAILED TO DOCK SUBSTRATE'
            sys.exit()
        self.best_score, self.best_pose = self.best(poses)
        self.substrate_score = self.best_score
        for inx, p in enumerate(poses):
            with EntryWriter(
                os.path.join(
                    self.settings.output_directory,
                    'gold_soln_CSD_API_m1_%d.mol2' % (inx+1)
                )) as w:
                w.write(self.best_pose)
        print '%-9s: %.3f' % ('Substrate', self.best_score)

    def best(self, poses):
        '''Extracts best score and best structure from the docked poses.'''
        fitness_tag = dict(
            plp='Gold.PLP.Fitness',
            chemscore='Gold.Chemscore.Fitness',
            goldscore='Gold.Goldscore.Fitness',
            asp='Gold.ASP.Fitness'
        )[self.settings.fitness_function]
        scores = [
            float(e.attributes[fitness_tag]) for e in poses
        ]
        best_score = max(scores)
        best_pose = poses[scores.index(best_score)]
        return best_score, best_pose

################################################################################

if __name__ == '__main__':
    runner = Runner()
    runner.run()
```

API documentation

IO API

Introduction

Module for reading and writing of molecules, crystals and database entries.

There are three types of readers: **`MoleculeReader`**, **`CrystalReader`** and **`EntryReader`**. The latter is used to read in database entries. The readers inherit functionality from the private base class **`_DatabaseReader`**.

Retrieving database entries from the CSD:

```
# Creating a CSD entry reader
csd_entry_reader = EntryReader('CSD')

# Create a CSD entry reader including any updates
directory = ccdc.io.csd_directory()
csd_and_updates = glob.glob(os.path.join(directory, '*.inf'))
csd_and_updates_reader = EntryReader(csd_and_updates)

# Similarly a set of in-house ASER databases may be adjoined to the CSD by constructing readers over
# a list of .inf files.

# Retrieve an entry based upon its index
first_csd_entry = csd_entry_reader[0]

# Access an entry/crystal/molecule based upon on its identifier
abebuf_entry = csd_entry_reader.entry('ABEBUF')
abebuf_crystal = csd_entry_reader.crystal('ABEBUF')
abebuf_molecule = csd_entry_reader.molecule('ABEBUF')

# Loop over all CSD entries
for entry in csd_entry_reader:
    print(entry.identifier)

# Loop over all the molecules
for mol in csd_entry_reader.molecules():
    print(mol.smiles)
```

Accessing molecules from a file:

```
# Creating a molecule reader
mol_reader = MoleculeReader('my_molecules.mol2')

# Retrieve a molecule based upon its index
first_molecule = mol_reader[0]

# Loop over all molecules
for mol in mol_reader:
    print(mol.smiles)
```

There are three types of writers: **`MoleculeWriter`**, **`CrystalWriter`** and **`EntryWriter`**. The latter can be used to write out sdf files with the entry's attributes dictionary formatted as SD tags. The writers inherit functionality from the private base class **`_DatabaseWriter`**.

Using a **`MoleculeWriter`** to write out a molecule:

```
with MoleculeWriter('abebuf.mol2') as mol_writer:
    mol_writer.write(abebuf_molecule)
```

Seealso

Descriptive IO documentation.

API

CSD location and version number

ccdc.io.**csd_directory** ()
Return the directory containing the CSD.

ccdc.io.**csd_version** ()
Return the version of the CSD in use.

Readers

class ccdc.io.**_DatabaseReader** (*fname*, *aser=''*)
Base class for database readers.
Readers are context managers, supporting the syntax:

```
with MoleculeReader(filename) as filehandle:
    for mol in filehandle:
        print(mol.smiles)
```

close ()
Close the database.

crystal (*id*)
Random access to crystals.

Parameters: **id** -- **ccdc.crystal.Crystal.identifier**
Returns: **ccdc.crystal.Crystal**

crystals ()
Generator for crystals in the database.

entries ()
Generator for entries in the database.

entry (*id*)
Random access to entries.

Parameters: **id** -- **ccdc.entry.Entry.identifier**
Returns: **ccdc.entry.Entry**

identifier (*i*)
Random access to identifiers.

Parameters: **i** -- int index
Returns: str identifier

journals
The list of journals held in a database.

molecule (*id*)
Random access to molecules

Parameters: **id** -- **ccdc.molecule.Molecule.identifier**

Returns: **ccdc.molecule.Molecule**

molecules ()
Generator for molecules of the database.

class ccdc.io.**EntryReader**
Treat the database as a source of entries.

An EntryReader can instantiated using:

- The explicit string 'CSD', which defaults to the CSD.
- A file name with an optional format argument. If the format argument is empty it uses the suffix of the file name to infer the file format.

One of the supported file formats is 'identifiers' in which case the file is assumed to contain a new line separated list of refcodes from the CSD. The suffix of such a file may be '.gcd'.
During initialisation a **_DatabaseReader** is dynamically bound to the **EntryReader** instance, which means that the methods of **_DatabaseReader** are available from the **EntryReader** instance.

```
>>> csd_entry_reader = EntryReader('CSD')
>>> type(csd_entry_reader[0])
<class 'ccdc.entry.Entry'>
>>> csd_entry_reader.identifier(0)
u'AABHTZ'
>>> aabhtz_entry = csd_entry_reader.entry('AABHTZ')
>>> aabhtz_entry.publication.authors
u'P.-E.Werner'
```

class ccdc.io.**CrystalReader**
Treat the database as a source of crystals.

A CrystalReader can be instantiated using:

- The explicit string 'CSD', which defaults to the CSD.
- A file name with an optional format argument. If the format argument is empty it uses the suffix of the file name to infer the file format.

One of the supported file formats is 'identifiers' in which case the file is assumed to contain a new line separated list of refcodes from the CSD. The suffix of such a file may be '.gcd'.
During initialisation a **_DatabaseReader** is dynamically bound to the **CrystalReader** instance, which means that the methods of **_DatabaseReader** are available from the **CrystalReader** instance.

```
>>> csd_crystal_reader = CrystalReader('CSD')
>>> type(csd_crystal_reader[0])
<class 'ccdc.crystal.Crystal'>
>>> csd_crystal_reader.identifier(0)
u'AABHTZ'
>>> aabhtz_crystal = csd_crystal_reader.crystal('AABHTZ')
>>> aabhtz_crystal.crystal_system
u'triclinic'
```

class ccdc.io.**MoleculeReader**
Treat the database as a source of molecules.

A MoleculeReader can be instantiated using:

- The explicit string 'CSD', which defaults to the CSD.
- A file name with an optional format argument. If the format argument is empty it uses the suffix of the file name to infer the file format.

One of the supported file formats is 'identifiers' in which case the file is assumed to contain a new line separated list of refcodes from the CSD. The suffix of such a file may be '.gcd'.

During initialisation a **_DatabaseReader** is dynamically bound to the **MoleculeReader** instance, which means that the methods of **_DatabaseReader** are available from the **CrystalReader** instance.

```
>>> csd_molecule_reader = MoleculeReader('CSD')
>>> type(csd_molecule_reader[0])
<class 'ccdc.molecule.Molecule'>
>>> csd_molecule_reader.identifier(0)
u'AABHTZ'
>>> aabhtz_molecule = csd_molecule_reader.molecule('AABHTZ')
>>> aabhtz_molecule.smiles
u'CC(=O)NN1C=NN=C1N(N=Cc1c(Cl)cccc1Cl)C(C)=O'
```

Writers

class ccdc.io.**_DatabaseWriter** (*fname*)

Base class for AserDatabase and other database formats.
Writers are context managers, supporting the syntax:

```
with MoleculeWriter('output.mol2') as filehandle:
    filehandle.write(mol)
```

close ()

Close the database.

remove ()

Remove the file if it exists.

write_crystal (*c*)

Appends an entry to the database to be written out.

Parameters: **c** -- **ccdc.crystal.Crystal**

write_entry (*e*)

Appends an entry to the database to be written out.

Parameters: **e** -- **ccdc.entry.Entry**

write_molecule (*m*)

Appends a molecule to the database to be written out.

Parameters: **m** -- **ccdc.molecule.Molecule**

class ccdc.io.**EntryWriter**

Writes Database Entries by default.

write (*e*)

Write the entry.

Parameters: **e** -- **ccdc.entry.Entry**

class ccdc.io.**CrystalWriter**

Writes crystals by default.

write (*c*)

Write the crystal.

Parameters: **c** -- **ccdc.crystal.Crystal**

class ccdc.io.**MoleculeWriter**

Writes molecules by default.

write (*m*)
Write the molecule.

Parameters: **m** -- **ccdc.molecule.Molecule**

Entry API

Introduction

The main class of the **ccdc.entry** module is **ccdc.entry.Entry**.

A **ccdc.entry.Entry** is often a CSD entry. It contains attributes that are beyond the concepts of chemistry and crystallography. An example of such an attribute would be the publication details of a CSD entry.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> abebuf = csd_reader.entry('ABEBUF')
>>> print(abebuf.publication) # doctest: +NORMALIZE_WHITESPACE
Citation(authors=u'S.W.Gordon-Wylie, E.Teplin, J.C.Morris, M.I.Trombley,
                  S.M.McCarthy, W.M.Cleaver, G.R.Clark',
         journal_name=u'Cryst.Growth Des.',
         volume=u'4', year=2004, first_page=u'789',
         doi=u'10.1021/cg049957u')
```

However, a **ccdc.entry.Entry** does not necessarily have to be a CSD entry. If, for example, a sdf file is read in using a **ccdc.io.EntryReader** then the sdf tags will be added to a dictionary-like object named `attributes` of the entry. Entries read in from cif files will also contain `attributes` with the raw data from the cif file.

Seealso

Descriptive entry documentation.

API

class ccdc.entry.**Entry** (*_entry=None*)
A database entry.

class **CrossReference** (*_xr*)
A cross-reference in the database.

identifiers
The identifiers of the cross-referenced structures.

scope
Whether the cross-reference applies to the individual identifier or the family of related identifiers.

text
The text of the cross-reference.

type
The type of the cross-reference.
This may be 'Racemate', 'Stereoisomer', 'Isomer', 'Reinterpretation of', 'Reinterpretation ref', 'Coordinates ref'

Entry.**analogue**
Analogue information.

Entry.**bioactivity**
Recorded information about bioactivity if available otherwise None.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> aadmpy10 = csd_reader.entry('AADMPY10')
>>> aadmpy10.bioactivity
u'antineoplastic activity'
```

Entry.**ccdc_number**
The CCDC deposition number.

```
>>> from ccdc.io import EntryReader
>>> entry_reader = EntryReader('CSD')
>>> abebuf = entry_reader.entry('ABEBUF')
>>> print(abebuf.ccdc_number)
241370
```

Entry.**chemical_name**
The chemical name of the entry.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> acsala13 = csd_reader.entry('ACSALA13')
>>> acsala13.chemical_name
u'Acetylsalicylic acid'
```

Entry.**color**
The colour of the crystal if given, otherwise None.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> acesor = csd_reader.entry('ACESOR')
>>> acesor.color
u'yellow'
```

Entry.**cross_references**
The tuple of **ccdc.entry.Entry.CrossReference** for this entry.

Entry.**crystal**
The **ccdc.crystal.Crystal** contained in a database entry.

Entry.**disorder_details**
Information about any disorder present if given otherwise None.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> abacir = csd_reader.entry('ABACIR')
>>> abacir.disorder_details
u'O4 and O4A disordered over two sites with occupancies 0.578:0.422.'
```

Entry.**disordered_molecule**
The **ccdc.molecule.Molecule** contained in a database entry, including disordered atoms.

Entry.**formula**

The published chemical formula in an entry.
If no published chemical formula is available it will be calculated from the molecule.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> aacani10 = csd_reader.entry('AACANI10')
>>> aacani10.formula
u'C10 H18 N2 Ni1 O5,2(H2 O1)'
```

static Entry.**from_molecule** (*mol*, ***attributes*)
Construct an entry from a molecule, using the keyword arguments as attributes.

static Entry.**from_string** (*s*, *format='mol2'*)
Create an entry from a string representation.

Parameters:
- **s** -- string representation of an entry, crystal or molecule
- **format** -- one of 'mol2', 'sdf', 'mol' or 'cif'

Returns: a **ccdc.entry.Entry**

Raises: TypeError for invalid format

Entry.**habit**
The crystal habit.

Entry.**has_3d_structure**
Whether the entry has 3d information.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> aqefor = csd_reader.entry('AQEFOR')
>>> aqefor.has_3d_structure
False
```

Entry.**has_disorder**
Whether the structure has disorder.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> abacir = csd_reader.entry('ABACIR')
>>> abacir.has_disorder
True
```

Entry.**identifier**
The string identifier of the entry, e.g. 'ABEBUF'.

Entry.**is_organic**
Whether the structure is organic.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> aacani10 = csd_reader.entry('AACANI10')
>>> aacani10.is_organic
False
```

Entry.**is_organometallic**
Whether the structure is organometallic.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> aacani10 = csd_reader.entry('AACANI10')
>>> aacani10.is_organometallic
True
```

Entry.**is_polymeric**
Whether the structure contains polymeric bonds.

```
>>> from ccdc.io import EntryReader
>>> csd = EntryReader('CSD')
>>> abacuf = csd.entry('ABACUF')
>>> abacuf.is_polymeric
True
```

Entry.**is_powder_study**
Whether or not the crystal determination was performed on a powder study

```
>>> from ccdc.io import EntryReader
>>> csd = EntryReader('csd')
>>> print(csd.entry('AABHTZ').is_powder_study)
False
>>> print(csd.entry('ACATAA').is_powder_study)
True
```

Entry.**melting_point**
Melting point of the crystal if given otherwise None.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> acsala13 = csd_reader.entry('ACSALA13')
>>> acsala13.melting_point
u'135.5deg.C'
```

Entry.**molecule**
The **ccdc.molecule.Molecule** contained in a database entry.

Entry.**peptide_sequence**
The peptide sequence of the entry if any.

Entry.**phase_transition**
Phase transition of the entry.

Entry.**polymorph**
Polymorphic information about the crystal if given otherwise None.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> acsala13 = csd_reader.entry('ACSALA13')
>>> acsala13.polymorph
u'polymorph II'
```

Entry.**pressure**
The experimental pressure of the crystallisation of the entry, where known.
This is a text field. If None the experiment was performed at ambient temperature or not recorded.

```
>>> from ccdc.io import EntryReader
>>> csd = EntryReader('csd')
>>> print(csd.entry('AABHTZ').pressure)
None
>>> print(csd.entry('ABULIT03').pressure)
at 1.4 GPa
```

Entry.**previous_identifier**
Previous identifier if any.

```
>>> from ccdc.io import EntryReader
>>> entry_reader = EntryReader('CSD')
>>> acpret03 = entry_reader.entry('ACPRET03')
>>> acpret03.identifier
u'ACPRET03'
>>> acpret03.previous_identifier
u'DABHUJ'
```

Entry.**publication**
The first publication of a structure.
Expressed as a tuple of (authors, journal, volume, year, first_page).

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> aqefor = csd_reader.entry('AQEFOR')
>>> aqefor.publication  # doctest: +NORMALIZE_WHITESPACE
Citation(authors=u'M.E.Bluhm, M.Ciesielski, H.Gorls, O.Walter, M.Doring',
         journal_name=u'Inorg.Chem.', volume=u'42', year=2003,
         first_page=u'8878', doi=u'10.1021/ic034773a')
```

Entry.**r_factor**
Resolution of crystallographic determination, given as a percentage value.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> acsala13 = csd_reader.entry('ACSALA13')
>>> acsala13.r_factor
16.22
```

Entry.**radiation_source**
The radiation source of the crystal's determination.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> aabhtz = csd_reader.entry('AABHTZ')
>>> aabhtz.radiation_source
'X-ray'
>>> csd_reader.entry('ABINOR01').radiation_source
'Neutron'
```

Entry.**remarks**
Any remarks on the entry registered by the editors of the database.
These may include details like a US Patent number:

```
>>> from ccdc.io import EntryReader
>>> csd = EntryReader('csd')
>>> print(csd.entry('ARISOK').remarks)
U.S. Patent 2005/6858644 B2
```

or reflect editorial decisions: >>> print(csd.entry('ABAPCU').remarks) The position of the hydrate is dubious. It has been deleted

`Entry.`**`solvent`**

Recrystallisation solvent.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> rechul = csd_reader.entry('REKHUL')
>>> rechul.solvent
u'pentane'
```

`Entry.`**`source`**

The source of the compound(s) in the entry.

```
>>> from ccdc.io import EntryReader
>>> entry_reader = EntryReader('CSD')
>>> fifdut = entry_reader.entry('FIFDUT')
>>> print(fifdut.source)
dried venom of Chinese toad Ch'an Su
```

`Entry.`**`synonyms`**

List containing any recorded synonyms for the entry.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> acsala13 = csd_reader.entry('ACSALA13')
>>> acsala13.synonyms
(u'Aspirin',)
```

`Entry.`**`temperature`**

Experimental temperature of the entry

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> acsala13 = csd_reader.entry('ACSALA13')
>>> acsala13.temperature
u'at 100 K'
```

`Entry.to_string (`*`format='mol2'`*`)`

Return a string representation of an entry.

Parameters: **format** -- 'mol2', 'sdf', 'mol' or 'cif'

Return type: string

Raises: TypeError if the format is not 'mol2', 'sdf', 'mol' or 'cif'

Crystal API

Introduction

The main class of the **ccdc.crystal** module is **ccdc.crystal.Crystal**.

A **ccdc.crystal.Crystal** contain attributes and functions that relate to crystallography. An example of a crystallographic attribute is the **ccdc.crystal.Crystal.cell_volume**.

```
>>> from ccdc.io import CrystalReader
>>> csd_crystal_reader = CrystalReader('CSD')
>>> first_csd_crystal = csd_crystal_reader[0]
>>> round(first_csd_crystal.cell_volume, 3)
769.978
```

Seealso

Descriptive crystal documentation.

API

class ccdc.crystal.**Crystal** (*crystal*, *identifier*)
Represents a crystal.

class **Contact** (*_view*, *_contact*)
A crystallographic contact.

intermolecular
Whether the contact is inter- or intra-molecular.

symmetry_operators
The symmetry operators by which the atoms of the contact are related.

type
The type of contact.

class Crystal.**HBond** (*_view*, *_contact*)
A crystallographic hydrogen bond.

intermolecular
Whether the contact is inter- or intra-molecular.

symmetry_operators
The symmetry operators by which the atoms of the contact are related.

type
The type of contact.

class Crystal.**ReducedCell** (*_cell*)
The reduced cell of a crystal.

cell_angles
The cell angles of the reduced cell.

cell_lengths
The cell lengths of the reduced cell.

volume
The volume of the reduced cell.

Crystal.assign_bonds ()

Detect and assign bond types in the crystal.
This function will ignore any existing bond information and will use the geometry of the crystal to detect and assign bond types.

`Crystal.`**`calculated_density`**
Calculated density of the crystal.

`Crystal.`**`cell_angles`**
Tuple containing the cell angles; (alpha, beta, gamma).
Note that the angles are reported in degrees. The returned value may be addressed by index or by key:

```
>>> a = CellAngles(90.0, 45.0, 33.3)
>>> a.alpha
90.0
>>> a[1]
45.0
```

`Crystal.`**`cell_lengths`**
Tuple containing the cell axis lengths: (a, b, c).
The returned value may be addressed by index or key:

```
>>> l = CellLengths(1, 2, 3)
>>> l[0]
1
>>> l.b
2
```

`Crystal.`**`cell_volume`**
Volume of the unit cell.

`Crystal.contacts (`*`intermolecular='Any'`*`, `*`distance_range=(-5.0`*`, `*`0.0)`*`, `*`vdw_corrected=True`*`, `*`path_length_range=(4`*`, `*`999)`*`)`
The collection of crystallographic contacts in this crystal structure.
A crystallographic contact may span two symmetry related molecules of the crystal.

Parameters:
- **intermolecular** -- Intramolecular, Intermolecular or Any.
- **distance_range** -- The minimum and maximum distances considered to be acceptable for a contact to be formed.
- **path_length_range** -- minimum and maximum values for the length of path between the contact atoms

Returns: a tuple of **`ccdc.crystal.Crystal.Contact`**

`Crystal.`**`crystal_system`**
The spacegroup system of the crystal.

`Crystal.`**`disordered_molecule`**
The underlying molecule with disordered atoms represented.

Raises: TypeError if the crystal has no atoms.

`Crystal.`**`formula`**
Return the chemical formula of the molecule in the crystal.

static `Crystal.`**`from_string`** (*`s`*, *`format='mol2'`*)
Create a crystal from a string representation.

Parameters:
- **s** -- string representation of a crystal or molecule
- **format** -- one of 'mol2', 'sdf', 'mol' or 'cif'

Returns: a **ccdc.crystal.Crystal**

Raises: TypeError for invalid format

Crystal.hbonds (*intermolecular='Any'*, *distance_range=(-5.0*, *0.0)*, *angle_tolerance=120.0*, *vdw_corrected=True*, *require_hydrogens=True*, *path_length_range=(4*, *999)*)

The collection of crystallographic hydrogen bonds in this crystal structure.
A crystallographic hydrogen bond may span two symmetry related molecules of the crystal.

Parameters:
- **intermolecular** -- Intramolecular, Intermolecular or Any.
- **distance_range** -- The minimum and maximum distances considered to be acceptable for a hydrogen bond to be formed.
- **vdw_corrected** -- Whether the distances are relative to the Van der Waals radius of the atoms.
- **angle_tolerance** -- the acceptable range for a hydrogen Donor-Hydrogen-Acceptor angle.
- **require_hydrogens** -- whether or not hydrogens will be necessary for the hydrogen bond.
- **path_length_range** -- minimum and maximum values for the length of path between the donor and acceptor atom

Returns: a tuple of **ccdc.crystal.Crystal.HBond**

Crystal.**identifier**

The string identifier of the crystal, e.g. 'ABEBUF'.

Crystal.**lattice_centring**

The lattice centring of this crystal.

Crystal.molecular_shell (*distance_type='actual'*, *distance_range=(0.0*, *3.0)*, *atom_selection=[*, *]*)

Return a molecule containing the atoms within a distance cutoff.
A subset of atoms to base the expansion on can be provided using the atom_selection argument.
If the distance type is VdW then the min to max range is relative to the sum of the vdW radii.

Parameters:
- **distance_type** -- 'vdw' or 'actual'.
- **distance_range** -- tuple containing the minimum and maximum for the contact distance range.
- **atom selection** -- list of atoms to base the expansion on.

Returns: **ccdc.molecule.Molecule** containg the atoms within the distance cutoff excluding any atoms from the molecules defined in the atom selection.

Crystal.**molecule**

The underlying molecule.
Note that a molecule can have several components.

Raises: TypeError if the crystal has no atoms

Crystal.**packing_coefficient**

The packing coefficient of the crystal.
Measures the proportion of the unit cell occupied by atoms. It is a fraction between zero and one; going from unoccupied to completely filled.

Crystal.packing_shell (*packing_shell_size=15*)

Create a packing shell of the crystal.

This method is available only to licensed users.

Parameters: **packing_shell_size** -- the required number of molecules in the packing shell

Returns: a **ccdc.molecule.Molecule** containing packing_shell_size replicas of the crystal.

Crystal.**reduced_cell**
The reduced cell of the crystal.

Crystal.**spacegroup_number_and_setting**
The number in international tables and setting of the crystal's spacegroup.

```
>>> from ccdc.io import CrystalReader
>>> csd_crystal_reader = CrystalReader('CSD')
>>> crystal = csd_crystal_reader.crystal('AABHTZ')
>>> crystal.spacegroup_number_and_setting
(2, 1)
```

Crystal.**spacegroup_symbol**
The spacegroup symbol of the crystal.

Crystal.symmetric_molecule (*symmop*, *translate*, *force=False*)
Generate a symmetry related copy of the molecule.
This method may be used to build multi-molecular crystals for visualisation and other purposes.

Parameters:

- **symmop** -- a string representation of the symmetry operation, such as '-x,1/2+y,1/2-z' representing a 2-fold screw axis.
- **translate** -- a sequence of three integers representing the translational component to be applied to the symmetry operation.
- **force** -- a boolean value to allow symmetry operators not supported by the crystal to be applied.

Returns: a **ccdc.molecule.Molecule** derived from this crystal with the symmetry operation applied.

Raises TypeError: if the symmetry operator is not in the symmetry operators of this crystal and force is not True.

Crystal.**symmetry_operators**
The symmetry operators pertaining to this crystal.

Returns: a tuple of string representations of the symmetry operators

static Crystal.**symmetry_rotation** (*operator*)
The rotational component of the symmetry operator.

Parameters: **operator** -- a string representation of a symmetry operator

Returns: a tuple of 9 integer values representing the 3x3 rotation matrix of the operator

static Crystal.**symmetry_translation** (*operator*)
The translational component of the symmetry operator.

Parameters: **operator** -- a string representation of a symmetry operator

Returns: a tuple of three floats representing the translational component of the operator

Crystal.to_string (*format='mol2'*)
Convert the crystal to a mol2 representation.

Parameters: **format** -- 'mol2', 'sdf' or 'cif'

Returns: string representation in the appropriate format

Raises: TypeError if format is not as above.

Crystal.void_volume (*probe_radius=1.2*, *grid_spacing=0.7*, *mode='contact'*)
Determine the void volume of the crystal.

```
>>> from ccdc.io import EntryReader
>>> entry_reader = EntryReader('CSD')
>>> abawop_crystal = entry_reader.crystal('ABAWOP')
>>> round(abawop_crystal.void_volume(), 2)
12.71
```

Parameters:
- **probe_radius** -- float, size of the probe
- **grid_spacing** -- float, fineness of the grid on which the calculation is made
- **mode** -- either 'accessible' or 'contact' according to whether the centre of the probe or the whole probe must be accomodated.

Returns: void volume as a percentage of the unit cell volume

Crystal.**z_prime**
The number of molecules in the asymmetric unit.

Crystal.**z_value**
The number of molecules in the unit cell.

class ccdc.crystal.**PackingSimilarity** (*settings=None*)
Compare crystal packing similarities.
The crystal packing similarity feature is available only to CSD-Materials and CSD-Enterprise users.

class **Comparison** (*_comp*, *reference*, *target*)
The result of a comparison.

nmatched_molecules
The number of molecules of the crystal matched.

overlay_molecules ()
Return the overlayed molecules.

Returns: a pair of packing shells, the first of which has been overlayed on the second

packing_shell_size
Number of molecules in the packing shell.

rmsd
RMSD of the reference and the target.

class PackingSimilarity.**Settings** (*_settings=None*)
Settings for Packing similarity.

allow_artificial_inversion
Whether or not to invert a structure where there is no inversion symmetry.

allow_molecular_differences
Whether different compounds may be compared.

angle_tolerance
Maximum difference between reference and target angles.

distance_tolerance
Maximum difference between reference and target distances as a decimal fraction.

ignore_bond_counts

Whether or not to take account of bond counts.

ignore_bond_types
Whether or not to take accound of bond types.

ignore_hydrogen_counts
Whether or not to ignore hydrogen counts.

ignore_hydrogen_positions
Whether or not H positions should be ignored.

ignore_smallest_components
Whether or not to take account of solvents.

match_entire_packing_shell
Whether or not all molecules of the shell have to be matched.

molecular_similarity_threshold
Do not compare structures whose similarity is lower than this value.

packing_shell_size
The number of molecules in the packing shell.

show_highest_similarity_result
Control number of results.
For structures with Z' > 1 there will be more than one result.

skip_when_identifiers_equal
Do not compare structures with the same identifier.

PackingSimilarity.compare (*reference*, *target*)
Compare two crystals.

Parameters:
- **reference** -- **ccdc.crystal.Crystal**
- **target** -- **ccdc.crystal.Crystal**

Returns: **ccdc.crystal.PackingSimilarity.Comparison**, a tuple of comparisons if there are more than one or None if no comparison was possible

Molecule API

Introduction

The **ccdc.molecule** contains classes concerned with chemistry.

The three main classes of the **ccdc.molecule** module are:

- **ccdc.molecule.Atom**
- **ccdc.molecule.Bond**
- **ccdc.molecule.Molecule**

The **ccdc.molecule.Molecule** class contains attributes relating to the chemistry of the molecule, for example the SMILES representation.

```
>>> from ccdc.io import MoleculeReader
>>> csd_mol_reader = MoleculeReader('CSD')
>>> first_mol_in_csd = csd_mol_reader[0]
>>> first_mol_in_csd.smiles
u'CC(=O)NN1C=NN=C1N(N=Cc1c(Cl)cccc1Cl)C(C)=O'
```

In some cases a molecule may not have a canonical SMILES representation - where the structure has unknown atoms or bonds. In these cases the value will be None:

```
>>> AJABIX01 = csd_mol_reader.molecule('AJABIX01')
>>> print(AJABIX01.smiles)
None
```

The **ccdc.molecule.Molecule** also contains lists of atoms and bonds.

```
>>> len(first_mol_in_csd.atoms)
35
>>> len(first_mol_in_csd.bonds)
36
>>> first_atom = first_mol_in_csd.atoms[0]
>>> first_bond = first_mol_in_csd.bonds[0]
```

The **ccdc.molecule.Atom** contains attributes such as the **ccdc.molecule.Atom.atomic_symbol**. >>> first_atom.atomic_symbol u'Cl'

The **ccdc.molecule.Bond** contains attributes such as the **ccdc.molecule.Bond.bond_type**.

```
>>> first_bond.bond_type  # doctest: +ELLIPSIS
<ccdc.molecule.BondType object at ...>
>>> str(first_bond.bond_type)
'Single'
>>> first_bond.bond_type == 1
True
```

Seealso

Descriptive molecule documentation.

API

class ccdc.molecule.**Atom** (*atomic_symbol=''*, *coordinates=None*, *label=''*, *formal_charge=0*, *_atom=None*)

Represents an atom.

atomic_number

The atomic number of the element represented.

```
>>> atom = Atom(atomic_symbol='N')
>>> print(atom.atomic_number)
7
```

atomic_symbol

The atomic symbol of the element represented.

```
>>> atom = Atom(atomic_symbol='N')
>>> print(atom.atomic_symbol)
N
```

bonds

The bonds which this atom forms

chirality
The R/S Chirality flag for this atom.
Returns one of '', 'R', 'S', 'Mixed', 'Error'.

coordinates
The x, y, z coordinates of the atoms in orthogonal space.

```
>>> atom = Atom(atomic_symbol='C', coordinates=(1.0, 2.0, 3.0))
>>> atom.coordinates
Coordinates(x=1.0, y=2.0, z=3.0)
```

The coordinates may be addressed by index or by key:

```
>>> atom.coordinates.x
1.0
>>> atom.coordinates[2]
3.0
```

Note that this function will return None if the atom does not have 3D coordinates.

```
>>> atom = Atom(atomic_symbol='C')
>>> print(atom.coordinates)
None
```

formal_charge
Formal charge on the atom.

```
>>> atom = Atom(atomic_symbol='Cl')
>>> atom.formal_charge
0
```

fractional_coordinates
The fractional coordinates of an atom.
The coordinates of the atom expressed in terms of the underlying crystal's unit cell. Where there is no underlying crystal, these will appear relative to the default unit cell, so orthogonal and fractional coordinates coincide.

index
The index of this atom in its parent molecule.
Note that if an atom is not part of a molecule this function will return None.

```
>>> atom = Atom(atomic_symbol='Fe')
>>> print(atom.index)
None
```

is_acceptor
Test whether or not the atom is a hydrogen bond acceptor.

```
>>> atom = Atom(atomic_symbol='C')
>>> atom.is_acceptor
False
```

is_chiral
Whether the atom is chiral.

```
>>> atom = Atom()
>>> atom.is_chiral
False
```

Returns True if the R/S Chirality flag for this atom is one of 'R', 'S', or 'Mixed'.

is_cyclic
Test whether the atom is part of a ring system.

```
>>> atom = Atom()
>>> atom.is_cyclic
False
```

is_donor
Test whether or not the atom is a hydrogen bond donor.

```
>>> atom = Atom(atomic_symbol='C')
>>> atom.is_donor
False
```

is_metal
Whether this atom is a metal.

```
>>> atom = Atom(atomic_symbol='C')
>>> atom.is_metal
False
>>> atom = Atom(atomic_symbol='Hg')
>>> atom.is_metal
True
```

is_spiro
Whether this is a spiro atom.

label
The label for the atom.

```
>>> atom = Atom(atomic_symbol='C', label='C1')
>>> print(atom.label)
C1
>>> atom = Atom(atomic_symbol='C')
>>> print(atom.label)
<BLANKLINE>
```

neighbours
Those atoms bonded to this one

rings
The collection of **ccdc.molecule.Ring** in which this atom lies.

sybyl_type
The Sybyl atom type of an atom.

vdw_radius
The Van der Waals radius of the atom.

class ccdc.molecule.**Bond** (*bond_type=None*, *_bond=None*)

A bond between two atoms of a molecule.

class **BondType** (*bond_type*)
Represent a bond type as either an integer or a string.

```
>>> bond_type = Bond.BondType(1)
>>> print(bond_type)
Single
>>> bond_type == 1
True
```

static **all_bond_types** ()
Return dictionary of all valid CSD bond types.

Bond.**atoms**
The two atoms which this bond connects.

Bond.**bond_type**
The bond type represented as a **ccdc.molecule.Bond.BondType**.

Bond.**ideal_bond_length**
The ideal bond length for a bond of this type connected these atoms.

Bond.**is_conjugated**
Whether the bond is conjugated.

Bond.**is_cyclic**
Test whether the bond is part of a ring system.

Bond.**is_rotatable**
Test whether the bond is rotatable.

Bond.**rings**
The collection of **ccdc.molecule.Ring** of which this bond is a part.

Bond.**sybyl_type**
The Sybyl bond type of this bond.

class ccdc.molecule.**Ring** (*atoms*, *_molecule=None*)
A ring of atoms in a molecule.

bonds
The bonds of this ring.

is_aromatic
Whether or not the ring is aromatic.

is_fully_conjugated
Whether each bond of the ring is conjugated.

is_fused
Whether the ring is fused with another.

class ccdc.molecule.**Molecule** (*identifier=''*, *_molecule=None*)
Represents a molecule

class **Contact** (*_contact*)
A contact between two molecules.

atoms

The atoms forming this contact.

intermolecular
Whether the contect be inter- or intra-molecular.

length
The length of the contact.

strength
The strength of this contact.

type
The type of contact.

class Molecule.**HBond** (*_contact*)
A hydrogen bond between atoms of a molecule.

angle
The donor-hydrogen-acceptor angle or None if the hydrogen is implicit.

atoms
The atoms forming this contact.
If a Hydrogen is present in the contact it will be the central atom. Either the donor or the acceptor may come first: which may be determined by inspecting the neighbours of the atoms, or by testing the attribute **ccdc.molecule.atom.is_donor**.

da_distance
The donor-acceptor distance of this hydrogen bond.

length
The length of the hydrogen bond.
This is the distance between the hydrogen and the acceptor, or None if the hydrogen is implicit.

Molecule.add_atom (*atom*)
Add an atom.

Parameters: **atom** -- **ccdc.molecule.Atom**
Returns: a copy of the atom which is now in this molecule

Molecule.add_atoms (*iterable*)
Add atoms, whether from another molecule or constructed *ab initio*.

Parameters: **iterable** -- any iterable of **ccdc.molecule.Atom**
Returns: a tuple of copies of the atoms in iterable

Molecule.add_bond (*bond_type*, *atom1*, *atom2*)
Add a bond between two atoms.

Parameters:
- **bond_type** -- **ccdc.molecule.Bond.BondType** instance
- **atom2** (*atom1,*) -- **ccdc.molecule.Atom** instances which must already be present in the molecule

Molecule.add_bonds (*iterable*)
Add several bonds to the molecule.

Parameters: **iterable** -- should be an iterable of triples, (**ccdc.molecule.Bond.BondType**, **ccdc.molecule.Atom**, **ccdc.molecule.Atom**) where the atoms must be in this molecule

Molecule.add_group (*mol*, *atom1*, *atom2*)

Add a group from another molecule.
All atoms and bonds of downstream of the pair atom1, atom2 will be added to the molecule. :parameter mol: **ccdc.molecule.Molecule** :parameter atom1 atom2: **ccdc.molecule.Atom** which must be in mol

Molecule.add_hydrogens (*mode='all'*)
Add hydrogen atoms to the molecule.

Parameters: **mode** -- 'all' to generate all hydrogens (throws away existing hydrogens) or 'missing' to generate hydrogens deemed to be missing.
Raises: RuntimeError if any heavy atom has no site.

Molecule.add_molecule (*molecule*)
Add a copy of all the atoms and bonds of molecule.

Parameters: **molecule** -- **ccdc.molecule.Molecule**

Molecule.**all_atoms_have_sites**
Whether all atoms have coordinates.

Molecule.apply_quaternion (*Q*)
Apply a Quaternion to the molecule, rotating it about an arbitrary axis.

Parameters: **Q** -- a sequence of 4 floats

Molecule.assign_bond_types (*which='All'*)
Assign bond types to the molecule.

Parameters: **which** -- may be 'all' or 'unknown'
Raises: ValueError if an unrecognised `which` parameter is provided

Molecule.atom (*label*)
The unique atom with this label.

Parameters: **label** -- a string
Raises: RuntimeError if there is not exactly one atom in the molecule with this label
Returns: **ccdc.molecule.Atom**

Molecule.**atoms**
List of the atoms in the molecule.

Molecule.bond (*label1*, *label2*)
The bond between the atoms uniquely so labelled.

Parameters: **label2** (*label1,*) -- strings
Raises: RuntimeError if the atom labels are not unique or the atoms are not bonded
Returns: **ccdc.molecule.Bond**

Molecule.**bonds**
List of the bonds in the molecule.

Molecule.centre_of_geometry ()
Geometric centre of the molecule.

Raises: RuntimeError if an atom has no coordinates

Molecule.change_group (*atom1*, *atom2*, *mol*, *atom3*, *atom4*)
Transfer a copy of a group of atoms and bonds.
All atoms downstream of atom1 - atom2 will be removed and copies of all atoms and bonds downstream of atom3 - atom4 will be added to the molecule. The geometry of the copied atoms will be translated and rotated such that the bond atom3 - atom4 is aligned with atom1 - atom2.
The bond connecting the groups will have the same type as that between atom1 and atom2.

Parameters:
- **atom2 atom3 atom4** (*atom1*) -- **ccdc.molecule.Atom**
- **mol** -- **ccdc.molecule.Molecule** in which atom3 and atom4 must be present

Raises: RuntimeError if any atom of mol is siteless, or if the bond atom3 - atom4 is cyclic.

Returns: a tuple of **ccdc.molecule.Atom** which are the newly added atoms of the molecule.

Molecule.**components**
List of the disconnected molecules with the molecule.

Molecule.contacts (*distance_range=(-5.0, 0.0), only_strongest=False, path_length_range=(4, 999)*)
The collection of short contacts in this molecule.

Parameters:
- **distance_range** -- the minimum and maximum values (Van der Waals corrected) for a short contact.
- **only_strongest** -- whether or not to return only the strongest contact made.
- **path_length_range** -- minimum and maximum values for the length of path between the contact atoms.

Molecule.copy ()
Return a deep copy of the molecule.

Molecule.**formal_charge**
The formal charge of the molecule represented as an integer.

Molecule.**formula**
Return the chemical formula of the molecule.

static Molecule.**from_string** (*s*, *format='mol2'*)
Create a molecule from a string representation.

Parameters:
- **s** -- molecule string representation
- **format** -- one of 'mol2', 'sdf', 'mol' or 'cif'

Return type: **ccdc.molecule.Molecule**

Raises: TypeError if the format string is not 'mol2', 'sdf', 'mol' or 'cif'.

Raises: RuntimeError if the string representation is incorrectly formatted

Molecule.fuse_rings (*atom1*, *atom2*, *mol*, *atom3*, *atom4*)
Fuse a copy of a ring system.
The bond between atom3 and atom4 will be superimposed on the bond between atom1 and atom2.

Parameters:
- **atom2** (*atom1*) -- **ccdc.molecule.Atom** within this molecule
- **mol** -- **ccdc.molecule.Molecule**
- **atom4** (*atom3*) -- **ccdc.molecule.Atom** which must be within mol

Raises: RuntimeError if bonds between atoms are not cyclic

Returns: a tuple of **ccdc.molecule.Atom** which are the newly added atoms of the molecule

Molecule.hbonds (*distance_range=(-5.0, 0.0), angle_tolerance=120.0, vdw_corrected=True, require_hydrogens=True, path_length_range=(4, 999)*)
The collection of molecular hydrogen bonds in this molecule.

Parameters:

- **distance_range** -- The minimum and maximum distances considered to be acceptable for a hydrogen bond to be formed.
- **vdw_corrected** -- Whether the distances are relative to the Van der Waals radius of the atoms.
- **angle_tolerance** -- the acceptable range for a hydrogen Donor-Hydrogen-Acceptor angle.
- **require_hydrogens** -- whether or not hydrogens will be necessary for the hydrogen bond.
- **path_length_range** -- minimum and maximum values for the length of path between the donor and acceptor atom

Returns: a tuple of **ccdc.molecule.Molecule.HBond**

If hydrogens are not required, they will not appear in the **ccdc.molecule.Molecule.HBond.atoms**.

Molecule.**heaviest_component**
The heaviest component, useful for stripping solvents.

Molecule.**heavy_atoms**
List of the heavy atoms in the molecule.

Molecule.**identifier**
The string identifier of the molecule, e.g. 'ABEBUF'.

Molecule.**is_3d**
Whether the molecule has 3d coordinates for all heavy atoms.

Molecule.**is_organic**
Uses CSD definitions for organometallic molecules.

Molecule.**is_organometallic**
Uses CSD definitions for organometallic molecules.

Molecule.**is_polymeric**
Whether the molecule contains polymeric bonds.

Molecule.kekulize ()
Convert the molecule to a Kekule representation, replacing all aromatic bonds by alternating single and double bonds.

Molecule.**largest_ring_size**
The size of the largest basic **ccdc.molecule.Ring** in the molecule.

Returns: the size of the largest basic **ccdc.molecule.Ring** or None if there are no rings in the molecule.

Molecule.**molecular_weight**
The molecular weight of the molecule.

Molecule.normalise_hydrogens ()
Normalise the hydrogen atom positions in the molecule.
This will normalise the position of hydrogen atoms attached to oxygen, nitrogen or carbon atoms to standard X-H distance values derived from statistical surveys of neutron diffraction data.
Note that this function will only modify the X-H distance. In other words the bond vector direction will be unaffected.

Molecule.normalise_labels ()
Ensure labels of atoms are unique.
Labels will be the atom's atomic symbol followed by an integer.

Molecule.remove_atom (*atom*)
Remove an atom from the molecule.

Parameters: **atom** -- **ccdc.molecule.Atom** which must be in the molecule.
Raises: RuntimeError if atom is not in the molecule.

Molecule.remove_atoms (*iterable*)
Remove several atoms from the molecule.

Parameters: **iterable** -- any iterable of **ccdc.molecule.Atom**
Raises: RuntimeError if any atom is not in the molecule.

Molecule.remove_bond (*bond*)
Remove a bond.

Parameters: **bond** -- **ccdc.molecule.Bond** which must be in the molecule
Raises: RuntimeError if the bond is not present

Molecule.remove_bonds (*iterable*)
Remove bonds.

Parameters: **iterable** -- any iterable of **ccdc.molecule.Bond**
Raises: RuntimeError if any bond is not in the molecule

Molecule.remove_group (*atom1*, *atom2*)
Remove the group of atoms and bonds downstream from atom1, atom2.

Parameters: **atom2** (*atom1*) -- **ccdc.molecule.Atom** which must be in the molecule.

Molecule.remove_hydrogens ()
Remove all hydrogen atoms.

Molecule.remove_unknown_atoms ()
Remove all atoms with an atomic number of zero (e.g. lone pairs, dummy atoms).

Molecule.**rings**
The collection of basic **ccdc.molecule.Ring** in the molecule.

Molecule.rotate (*xyz*, *theta*)
Rotate a molecule about the vector xyz by theta degrees.
The molecule will be rotated about its centre of geometry.

Parameters:
- **xyz** -- the rotation axis, a sequence of length 3
- **theta** -- angle by which to rotate, in degrees

Molecule.set_bond_length (*atom1*, *atom2*, *distance*)
Set the bond length between the two atoms to distance.

Parameters:
- **atom2** (*atom1,*) -- **ccdc.molecule.Atom**
- **distance** -- the desired distance between atom1 and atom2

Raises: RuntimeError if the two atoms are not neighbours or if the bond between them is cyclic or if any affected atoms no coordinates

Molecule.set_coordinates (*mol*, *atoms=None*)
Set the coordinates of this molecule to those of the other molecule.
For example AACFAZ has no coordinates, so in some circumstances one might wish to use the coordinates from AACFAZ10.

Parameters:
- **mol** -- another molecule
- **atoms** -- an iterable of pairs of atoms, the first of which is in this molecule, the second in the other molecule. If this is empty (or None) then it is assumed that the molecules' atoms match exactly.

Molecule.set_formal_charges ()
Calculate formal charges for atoms assuming bond types and protonation are correct.

Molecule.set_torsion_angle (*atom1*, *atom2*, *atom3*, *atom4*, *theta*)
Set the torsion angle between the specified atoms to theta.

Parameters:
- **atom2, atom3, atom4** (*atom1,*) -- **ccdc.molecule.Atom**
- **theta** -- the desired torsion angle in degrees

Raises: RuntimeError if the atoms are not connected or if a bond is cyclic or if an affected atom has no coordinates.

Molecule.set_valence_angle (*atom1*, *atom2*, *atom3*, *theta*)
Set the valence angle subtended by atom1, atom2, atom3 to theta.

Parameters:
- **atom2, atom3** (*atom1,*) -- **ccdc.molecule.Atom**
- **theta** -- the desired angle in degrees

Raises: RuntimeError if the atoms are not connected, if either bond is cyclic, or if an affected atom has no coordinates.

Molecule.shortest_path (*atom1*, *atom2*)
Return the shortest path between two atoms.

Parameters:
- **atom1** -- **ccdc.molecule.Atom**
- **atom2** -- **ccdc.molecule.Atom**

Returns: integer of shortest path, or 0 if no path can be found.

Raises: TypeError if atoms are not in this molecule.

Molecule.shortest_path_atoms (*atom1*, *atom2*)
The list of atoms along the shortest path from atom1 to atom2.

Parameters:
- **atom1** -- **ccdc.molecule.Atom**
- **atom2** -- **ccdc.molecule.Atom**

Returns: list of **ccdc.molecule.Atom** instances

Raises: TypeError if atoms are not in this molecule.

Molecule.shortest_path_bonds (*atom1*, *atom2*)
The list of bonds along the shortest path from atom1 to atom2.

Parameters:
- **atom1** -- **ccdc.molecule.Atom**
- **atom2** -- **ccdc.molecule.Atom**

Returns: list of **ccdc.molecule.Bond** instances

Raises: TypeError if atoms are not in this molecule.

Molecule.**smallest_ring_size**
Size of the smallest **ccdc.molecule.Ring** in the molecule.

Returns: the size of the smallest **ccdc.molecule.Ring** or None if there are no rings in the molecule.

`Molecule.`**`smiles`**
The canonical SMILES representation of the molecule.
This will raise a RuntimeError if the molecule contains unknown atoms or bonds, delocalised bonds or if there are non-terminal hydrogens.

`Molecule.standardise_aromatic_bonds ()`
Standardise aromatic bonds to CSD conventions.

`Molecule.standardise_delocalised_bonds ()`
Standardise delocalised bonds to CSD conventions.

`Molecule.to_string (`*`format='mol2'`*`)`
Return a string representation of a molecule.

Parameters: **format** -- 'mol2', 'sdf', 'mol' or 'cif'
Return type: string
Raises: TypeError if the format string is not 'mol2', 'sdf', 'mol' or 'cif'.

`Molecule.transform (`*`M`*`)`
Apply a matrix to the coordinates of the molecule.
The method will not check that the matrix is appropriate for a molecular transformation; it is for the user to ensure that inappropriate affine transformations are avoided.

Parameters: **M** -- a sequence of length 4 of sequences of length 4 of floats

`Molecule.translate (`*`xyz`*`)`
Translate the whole molecule.

Parameters: **xyz** -- a sequence of three floats
Raises: RuntimeError if an atom has no coordinates

Search API

Introduction

The **`ccdc.search`** module provides various search classes.

The three main classes of the **`ccdc.search`** module are:

- **`ccdc.search.SubstructureSearch`**
- **`ccdc.search.SimilaritySearch`**
- **`ccdc.search.TextNumericSearch`**
- **`ccdc.search.ReducedCellSearch`**

These all inherit from the base class **`ccdc.search.Search`**. The base **`ccdc.search.Search`** contains nested classes defining basic search hits and settings:

- **`ccdc.search.Search.SearchHit`**
- **`ccdc.search.Search.Settings`**

The base class **`ccdc.search.Search`** also contains the **`ccdc.search.Search.search()`** function which is used to search the CSD.

All the searches except **`ccdc.search.TextNumericSearch`** also support searching of all of the below:

- a Python list of identifiers
- a molecule file path
- a **`ccdc.io`** reader
- an individual **`ccdc.molecule.Molecule`**

- an individual **ccdc.crystal.Crystal**
- a list of molecules, crystals or entries

The **ccdc.search.TextNumericSearch** can only sensibly be applied to the CSD.

The **ccdc.search.Search.search()** returns a list of **ccdc.search.Search.SearchHit** instances. Some of the searches make use of more specific search hit classes, namely:

- **ccdc.search.SubstructureSearch.SubstructureHit**
- **ccdc.search.SimilaritySearch.SimilarityHit**
- **ccdc.search.TextNumericSearch.TextNumericHit**

Most of the searches return simple Python lists of search hits. However, a search carried out using a **ccdc.search.SubstructureSearch** returns a **ccdc.search.SubstructureSearch.SubstructureHitList**, which contains a **ccdc.search.SubstructureSearch.SubstructureHitList.superimpose()** function for superimposing all the hits on the first instance in the list.

To illustrate some of the searches let us first get an aspirin molecule.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> mol = csd_reader.molecule('ACSALA')
```

Substructure searching.

```
>>> from ccdc.search import MoleculeSubstructure, SubstructureSearch
>>> substructure = MoleculeSubstructure(mol)
>>> substructure_search = SubstructureSearch()
>>> substructure_search.add_substructure(substructure)
0
>>> hits = substructure_search.search()
>>> len(hits)
34
```

Similarity searching.

```
>>> from ccdc.search import SimilaritySearch
>>> similarity_search = SimilaritySearch(mol)
>>> hits = similarity_search.search()
>>> len(hits)
66
```

Text numeric searching.

```
>>> from ccdc.search import TextNumericSearch
>>> text_numeric_search = TextNumericSearch()
>>> text_numeric_search.add_compound_name('aspirin')
>>> hits = text_numeric_search.search()
>>> len(hits)
58
```

Seealso

The descriptive documentation for the *general philosophy of searching*, *substructure searching*, *similarity searching*, *text numeric searching*, and *reduced cell searching*.

API

Classes for defining substructures

class ccdc.search.**QueryAtom** (*atomic_symbol=''*, *_substructure_atom=None*)

Atom used to define a substructure search.

A QueryAtom can be used to represent a single atom type or a set of atom types. A QueryAtom can also have additional constraints imposed on it, for example that it should be aromatic.

Let us create a query atom representing an oxygen atom.

```
>>> query_atom = QueryAtom('O')
>>> print(query_atom)
QueryAtom(O)
```

Suppose that we wanted the query atom to be either a carbon or a nitrogen atom.

```
>>> query_atom = QueryAtom(['C', 'N'])
>>> print(query_atom)
QueryAtom(C, N)
```

It is possible to add further constraints on a QueryAtom. For, example, we can insist that it should be aromatic.

```
>>> query_atom.aromatic = True
>>> print(query_atom.aromatic)
AtomAromaticConstraint: 1
>>> print(query_atom)
QueryAtom(C, N)[atom aromaticity: equal to 1]
```

acceptor

Constraint specifying whether or not the QueryAtom is an acceptor.

add_connected_element_count (*atomic_symbols*, *count*)

Set the number of connected elements constraint.

Constraint to define the number of times the QueryAtom should be connected to atoms with elements defined in the atomic_symbols list.

Parameters:

- **atomic_symbols** -- atomic symbol or list of atomic symbols.
- **count** -- integer or pair of integers specifying the number of, or range of, connections required.

aromatic

Constraint specifying whether or not the QueryAtom is aromatic.

cyclic

Constraint specifying whether or not the QueryAtom is part of a cycle.

cyclic_bonds

Constraint specifying the number of cyclic bonds of the QueryAtom.

donor

Constraint specifying whether or not the QueryAtom is a donor.

formal_charge

Constraint specifying the formal charge on the QueryAtom.

formal_valency

Constraint specifying the formal valency of the QueryAtom.

index

Index of this atom in a substructure.

num_bonds
Constraint specifying the number of bonds the QueryAtom may have.

num_hydrogens
Constraint specifying the number of hydrogens the QueryAtom may have.

smallest_ring
Constraint specifying the size of the smallest ring the QueryAtom forms part of.

unfused_unbridged_ring
Constraint specifying whether or not the QueryAtom is part of an unfused and unbridged ring.

class ccdc.search.**QueryBond** (*bond_type=None*, *_substructure_bond=None*)
Bond used to define a substructure search.
A QueryBond can be used to represent a single bond type or a set of bond types. A QueryBond can also have additional constraints imposed on it, for example that it should be cyclic.
Let us create a QueryBond that will match any bond type.

```
>>> query_bond = QueryBond()
>>> print(query_bond)  # doctest: +NORMALIZE_WHITESPACE
QueryBond(Unknown, Single, Double, Triple,
          Quadruple, Aromatic, Delocalised, Pi)
```

To create a more specific QueryBond we need to specify some bond types.

```
>>> from ccdc.molecule import Bond
>>> single_bond = Bond.BondType('Single')
>>> double_bond = Bond.BondType('Double')
>>> query_bond = QueryBond(single_bond)
>>> print(query_bond)
QueryBond(Single)
>>> query_bond = QueryBond([single_bond, double_bond])
>>> print(query_bond)  # doctest: +NORMALIZE_WHITESPACE
QueryBond(Single, Double)
```

Finally, let us set a constraint for the bond to be cyclic.

```
>>> query_bond.cyclic = True
>>> print(query_bond)
QueryBond(Single, Double)[bond cyclicity: equal to 1]
```

```
>>> print(query_bond.cyclic)
BondCyclicityConstraint: 1
```

atoms
A list of the two QueryAtoms of the bond, if it is in a substructure, or None.

bond_length
Constraint specifying the length of the bond.

bond_polymeric
Constraint specifying whether or not the **QueryBond** is polymeric.

bond_unfused_unbridged_ring
Constraint specifying whether or not the **QueryBond** is part of an unfused and unbridged ring.

cyclic
Constraint specifying whether or not the **QueryBond** is part of a cycle.

class ccdc.search.**QuerySubstructure** (*_substructure=None*)

Class to define and run substructure searches.

As an example let us set up a QuerySubstructure for a carbonyl (C=O).

```
>>> from ccdc.molecule import Bond
>>> double_bond = Bond.BondType('Double')
>>> substructure_query = QuerySubstructure()
>>> query_atom1 = substructure_query.add_atom('C')
>>> query_atom2 = substructure_query.add_atom('O')
>>> query_bond = substructure_query.add_bond(double_bond, query_atom1, query_atom2)
```

add_atom (*atom*)

Add an atom to the substructure.

Parameters: **atom** -- may be a QueryAtom separately constructed, an atom of a molecule, or an atomic symbol.

Returns: **QueryAtom**

add_bond (*bond*, *atom1=None*, *atom2=None*)

Add a bond to the substructure.

Parameters:

- **bond** -- may be a **QueryBond**, a **ccdc.molecule.Bond.BondType**, a **ccdc.molecule.Bond**, a string or an int.
- **atom1** -- **QueryAtom** or None for any atom
- **atom2** -- **QueryAtom** or None for any atom

Returns: **QueryBond**

Raises: TypeError if an improper bond argument is supplied

atoms

The query atoms in the substructure.

bonds

The bonds in the substructure.

clear ()

Restart the query.

write_xml (*fname*)

Write an XML representation of the substructure.

Parameters: **fname** -- path to XML file

class ccdc.search.**SMARTSSubstructure** (*smarts*)

Make a substructure from a SMARTS string.

Let us create a ketone SMARTSSubstructure as an example.

```
>>> smarts_query = SMARTSSubstructure("[CD4][CD3](=[OD1])[CD4]")
>>> print(smarts_query.smarts)
[CD4][CD3](=[OD1])[CD4]
```

label_to_atom_index (*i*)

Translate a SMARTS label into the appropriate substructure atom index

smarts

The SMARTS string.

class ccdc.search.**MoleculeSubstructure** (*mol*)

Make a substructure query from an entire molecule.
Can be used to search for exact matches of a molecule. Furthermore if hydrogen atoms have been removed from the molecule used to initialise the MoleculeSubstructure it can be used to find hits that exactly match the heavy atoms. Note that molecules with more than one component will raise a TypeError, since multi-component molecule substructure searches are not supported. The components should be added as separate substructures.

class ccdc.search.**ConnserSubstructure** (*file_name*)
Read a Conquest query language file.

static **from_string** (*text*)
Create a substructure from a textual representation of a Connser file.

Search classes

class ccdc.search.**Search** (*settings=None*)
Common base class for searches

class **SearchHit** (*identifier*, *_database=None*, *_entry=None*, *_crystal=None*, *_molecule=None*, *_binary_database=None*)
Base class for search hits.
Provides access to molecules, crystals and entries.

crystal
The crystal corresponding to a search hit.

entry
The entry corresponding to a search hit.

molecule
The molecule corresponding to a search hit.

class Search.**Settings** (*_settings=None*)
Base class for search settings.

has_3d_coordinates
Constrain hits to have 3d coordinates.

max_hit_structures
The number of structures which may be returned from a search.

max_r_factor
Constrain the hits to have an R-factor less than this.

must_have_elements
Elements which must be present in a hit.

must_not_have_elements
Elements which must not be present in a hit.

no_disorder
Constrain hits to have no disorder.
The value will be False (no filtering), 'Non-hydrogen' (filter structures with heavy atom disorder) or 'All' (filter structures with any disordered atoms).

no_errors
Constrain the hits to have no suppressed errors.

no_ions
Constrain the hits not to have an atom with a formal charge.

no_metals
Constrain the hits not have a metal atom.

no_powder
Constrain hits not to be powder studies.

not_polymeric
Constrain the hits not to be polymeric structures.

only_organic
Constrain hits to be organic compounds.

only_organometallic
Constrain hits to be only organometallic compounds.

test (*argument*)
Test that the argument satisifies the requirements of the settings instance.

Parameters: **argument** -- a **ccdc.entry.Entry**, **ccdc.crystal.Crystal** or **ccdc.molecule.Molecule** instance.

Returns: bool

Search.search (*database=None*, *max_hit_structures=None*, *max_hits_per_structure=None*)
Perform a search.

class ccdc.search.**SimilaritySearch** (*mol=None*, *threshold=0.7*, *coefficient='tanimoto'*, *settings=None*)
Class to define and run similarity searches.

class **Screen** (*_fingers*)
Similarity screens.
A class for screening a reader to speed up similarity searches. The construction of the screens is computationally intensive, so is only worth doing if the same set of structures will be frequently searched. In this case it is important to write the screens, amortising the cost of construction over all the searches performed.
If a reader has a 'similarity_screen' attribute of this type, it will be used during a similarity search.

static **create** (*reader*)
Create screens for a reader.

static **from_file** (*file_name*)
Create screens from a file.

write (*file_name*)
Write the screens to a file.

class SimilaritySearch.**SimilarityHit** (*similarity*, *identifier*, *_database=None*, *_entry=None*, *_crystal=None*, *_molecule=None*, *_binary_database=None*)
A search hit recording the similarity measure.

SimilaritySearch.**coefficient**
Which coefficient to use when determining similarity.

static SimilaritySearch.**from_xml** (*xml*)
Create a SimilaritySearch from an XML representation.

Parameters: **xml** -- XML string

static SimilaritySearch.**from_xml_file** (*file_name*)
Create a SimilaritySearch from an XML file.

Parameters: **file_name** -- path to XML file

SimilaritySearch.read_xml (*xml*)

Read a query from an an XML representation.

Parameters: **xml** -- XML string

SimilaritySearch.read_xml_file (*file_name*)
Read an XML file into the similarity searcher.

Parameters: **file_name** -- path to XML file
Raises: IOError if the file cannot be read

SimilaritySearch.search_molecule (*mol*)
Search a molecule.
This can be used to determine a similarity coefficient against the given molecule.

Parameters: **mol** -- **ccdc.molecule.Molecule**
Returns: **SimilaritySearch.SimilarityHit**

SimilaritySearch.**threshold**
The similarity threshold to use.

class ccdc.search.**TextNumericSearch**
Class to define and run text/numeric searches in the CSD.
It is possible to add one or more criterion for the query to match.

```
>>> text_numeric_query = TextNumericSearch()
>>> text_numeric_query.add_compound_name('aspirin')
>>> text_numeric_query.add_citation(year=[2011, 2013])
>>> for hit in text_numeric_query.search(max_hit_structures=3):
...     print(hit.identifier)
...
ACSALA19
ACSALA20
ACSALA21
```

A human-readable representation of the queries may be obtained: >>> print(text_numeric_query.queries) (u'Compound name aspirin anywhere ', u'Journal year in range 2011-2013')

class **TextNumericHit** (*identifier*, *_db*)
Hit from a TextNumericSearch.

class TextNumericSearch.**TextNumericSearchSettings** (*_settings=None*)
Could put the modes in here as well? Definitely need to honour the max_hit_structures

TextNumericSearch.add_all_identifiers (*refcode*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for an identifier, including previous identifiers.

```
>>> from ccdc.search import TextNumericSearch
>>> query = TextNumericSearch()
>>> query.add_all_identifiers('DABHUJ')
>>> hits = query.search()
>>> print(hits[0].identifier)
ACPRET03
>>> print(hits[0].entry.previous_identifier)
DABHUJ
```

TextNumericSearch.add_all_text (*txt*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for text anywhere in the entry.

TextNumericSearch.add_analogue (*analogue*, *mode='anywhere'*, *ignore_non_alpha_num=False*)

Search for an analogue.

TextNumericSearch.add_author (*author*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for an author.

TextNumericSearch.add_bioactivity (*activity*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a particular bio-activity.

TextNumericSearch.add_ccdc_number (*value*)
Search for a particular or a range of CCDC deposition numbers.

```
>>> from ccdc.search import TextNumericSearch
>>> searcher = TextNumericSearch()
>>> searcher.add_ccdc_number(241370)
>>> hits = searcher.search()
>>> len(hits)
1
>>> entry = hits[0].entry
>>> print('%s %s' % (entry.identifier, entry.ccdc_number))
ABEBUF 241370
>>> searcher.clear()
>>> searcher.add_ccdc_number((241368, 241372))
>>> hits = searcher.search()
>>> print(len(hits))
3
>>> for hit in hits:
...     print('%s %s' % (hit.identifier, hit.entry.ccdc_number))
...
ABEBUF 241370
BIBZIW 241371
BIMGEK 241372
```

TextNumericSearch.add_citation (*author=''*, *journal=''*, *volume=None*, *year=None*, *first_page=None*, *ignore_non_alpha_num=False*, *_coden=None*)
Search for a citation.

TextNumericSearch.add_color (*color*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a particular colour.

TextNumericSearch.add_compound_name (*compound_name*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a compound name.
The search checks the content both of **ccdc.entry.Entry.chemical_name** and **ccdc.entry.Entry.synonyms**.
To illustrate this let us have a look at the CSD entry ABABEM.

```
>>> from ccdc.io import EntryReader
>>> entry_reader = EntryReader('CSD')
>>> ababem = entry_reader.entry('ABABEM')
>>> print(ababem.chemical_name)
Tetrahydro[1,3,4]thiadiazolo[3,4-a]pyridazine-1,3-dione
>>> print(ababem.synonyms[0])
8-Thia-1,6-diazabicyclo[4.3.0]nonane-7,9-dione
```

The text `azabicyclo[4.3.0]nonane` is only found in the synonym. Let us search for it using a compound name search.

```
>>> from ccdc.search import TextNumericSearch
>>> query = TextNumericSearch()
>>> query.add_compound_name('azabicyclo[4.3.0]nonane')
>>> hits = query.search()
```

Finally let us assert that we have found ABABEM.

```
>>> assert(u'ABABEM' in [h.identifier for h in hits])
```

TextNumericSearch.add_disorder (*disorder*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a disorder comment.

TextNumericSearch.add_doi (*doi*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a DOI.

TextNumericSearch.add_habit (*habit*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a particular habit.

TextNumericSearch.add_identifier (*refcode*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a refcode.

TextNumericSearch.add_peptide_sequence (*peptide_sequence*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a peptide sequence.

TextNumericSearch.add_phase_transition (*phase_transition*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a phase transition.

TextNumericSearch.add_polymorph (*polymorph*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for polymorph information.

TextNumericSearch.add_solvent (*solvent*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a solvent.

TextNumericSearch.add_source (*source*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a source.

```
>>> from ccdc.search import TextNumericSearch
>>> searcher = TextNumericSearch()
>>> searcher.add_source('toad')
>>> hits = searcher.search(max_hit_structures=5)
>>> for h in hits:
...     print('%-8s: %s' % (h.identifier, h.entry.source))
...
CUXYAV  : Ch'an Su (dried venom of Chinese toad)
FIFDUT  : dried venom of Chinese toad Ch'an Su
FIFFAB  : dried venom of Chinese toad Ch'an Su
FIFFAB01: dried venom of Chinese toad Ch'an Su
GIHWEZ  : Dried venom of Chinese toad Ch'an Su
```

TextNumericSearch.add_synonym (*synonym*, *mode='anywhere'*, *ignore_non_alpha_num=False*)
Search for a synonym.

TextNumericSearch.clear ()
Restart a search.

static TextNumericSearch.**from_xml** (*xml*)
Create a TextNumericSearch from XML.

Parameters: **xml** -- XML string

static TextNumericSearch.**from_xml_file** (*file_name*)
Create a TextNumericSearch from an XML file.

Parameters: **file_name** -- path to XML file

TextNumericSearch.is_journal_valid (*journal*)
Check the validity of a journal name.

TextNumericSearch.**journals**
A dictionary of journal name : ccdc code number.

TextNumericSearch.**queries**
The current set of queries for this search.

TextNumericSearch.read_xml (*xml*)
Read a query from XML.

Parameters: **xml** -- XML string

TextNumericSearch.read_xml_file (*file_name*)
Read a text numeric search from an XML file.

Parameters: **file_name** -- path to XML file
Raises: IOError if the file cannot be read

class ccdc.search.**SubstructureSearch** (*settings=None*)
Query crystal structures for interactions.

class **Screen** (*_screens*, *_fingerprints*)
Substructure screens.
A class for screening a reader to speed up substructure searches. The construction of the screens is computationally intensive, so is only worth doing if the same set of structures will be frequently searched. In this case it is important to write the screens, amortising the cost of construction over all the searches performed.
If a reader has a 'substructure_screen' attribute of this type, it will be used during a substructure search.

candidates (*search*)
Identify which structures are capable of being matched by a search.

Parameters: **search** -- **ccdc.search.SubstructureSearch** instance.
Returns: list of indices of structures which might match the search.

static **create** (*reader*)
Create screens for a reader.

static **from_file** (*file_name*)
Create screens from a file.

write (*file_name*)
Write the screens to a file.

class SubstructureSearch.**Settings** (*max_hit_structures=None*, *max_hits_per_structure=None*)
Settings appropriate to a substructure search.

max_hits_per_structure
Maximum number of hits per structure.

class SubstructureSearch.**SubstructureHit** (*identifier*, *match=None*, *search_structure=None*, *query=None*, *_database=None*, *_entry=None*, *_crystal=None*, *_molecule=None*, *_binary_database=None*)
A hit from an interaction search.

match_atoms (*indices=False*)
Return the atoms matched by the substructure.

Parameters: **indices** -- whether or not to return the atom indices instead of the **ccdc.molecule.Atom** instances

Returns: list of **ccdc.molecule.Atom** instances or atom indices

match_components ()
Return the molecular components matched by the search.

Returns: list of **ccdc.molecule.Molecule**

match_symmetry_operators ()
The symmetry operators required to form the match.

Returns: a list of symmetry operators in the order of the matched atoms.

class SubstructureSearch.**SubstructureHitList**
List of hits from an SubstructureSearch

superimpose ()
Superimpose all matched molecules on their query atoms
Just superimpose on first substructure

write_c2m_file (*file_name*)
Write a ConQuest to Mercury interchange file.

Parameters: **file_name** -- the file to which the data will be written.

This file will allow substructure search results to be read into the data analysis package of Mercury.

SubstructureSearch.add_angle_constraint (*name*, **args*)
Add a general (not necessarily valence) angle constraint.

Parameters:
- **name** -- string representing the name of the constraint
- **args** -- each arg is either a pair (substructure_id, atom_id) or a string denoting a previously defined geometric object, or alternately substructure_id followed by atom_id. The last argument is an angle range, (lower, upper).

SubstructureSearch.add_angle_measurement (*name*, **args*)
Measure an angle in any hit.

Parameters:
- **name** -- string representing the name of the measurement
- **args** -- either a sequence of substructure identifiers alternating with atom identifiers, or a sequence of pairs (substructure_id, atom_id) or a mixture of them. Any argument may be a string indicating a previously defined geometric object.

SubstructureSearch.add_centroid (*name*, **args*)
Adds a geometric centroid to the substructure search.
This may be used to define further measurements and constraints.

Parameters:
- **name** -- the name of the geometric object.
- **args** -- an even-numbered tuple of indices alternating substructure indices and atom indices, or a tuple of (substructure_id, atom_id) pairs.

SubstructureSearch.add_distance_constraint (*name*, **args*, ***kw*)
Add a distance constraint.

Parameters:
- **name** -- string representing the name of the constraint
- **args** -- each arg is either a pair (substructure_id, atom_id) or a string representing a previously define geometric object or a substructure_id followed by an atom_id. The last argument is a distance range (lower, higher).
- **kw** -- either type='Intermolecular', 'Intramolecular' or 'Any', and vdw_corrected=True or False.

SubstructureSearch.add_distance_measurement (*name*, **args*)
Measure a distance in any hit.

Parameters:
- **name** -- string representing the name of the measurement
- **args** -- either a sequence of substructure identifiers alternating with atom identifiers, or a sequence of pairs (substructure_id, atom_id) or a mixture of them. Any argument may be a string indicating a previously defined geometric object.

SubstructureSearch.add_plane (*name*, **args*)
Adds a geometric plane to the substructure search.
This may be used to define further measurements and constraints.

Parameters:
- **name** -- the name of the geometric object.
- **args** -- an even-numbered tuple of indices alternating substructure indices and atom indices, or a tuple of (substructure_id, atom_id) pairs.

SubstructureSearch.add_plane_angle_constraint (*name*, *plane1*, *plane2*, *c*)
Add a plane angle constraint.

Parameters:
- **name** -- the name of this constraint.
- **plane2** (*plane1,*) -- the names of two previously added planes.
- **c** -- the range of angles permitted in the structure.

SubstructureSearch.add_plane_angle_measurement (*name*, *plane1*, *plane2*)
Measure an angle between two planes.

Parameters:
- **name** -- string representing the name of the measurement.
- **plane2** (*plane1,*) -- the names of previously added planes.

SubstructureSearch.add_substructure (*substructure*)
Add a substructure.

Parameters: **substructure** -- **ccdc.search.QuerySubstructure**

SubstructureSearch.add_torsion_angle_constraint (*name*, **args*)
Add a torsion angle constraint.

Parameters:
- **name** -- string representing the name of the constraint
- **args** -- each arg is either a pair (substructure_id, atom_id) or a string denoting a previously defined geometric object, or alternately substructure_id followed by atom_id. The last argument is an angle range, (lower, upper).

SubstructureSearch.add_torsion_angle_measurement (*name*, **args*)
Measure a torsion angle in any hit.

Parameters:

- **name** -- string representing the name of the measurement
- **args** -- either a sequence of substructure identifiers alternating with atom identifiers, or a sequence of pairs (substructure_id, atom_id) or a mixture of them. Any argument may be a string indicating a previously defined geometric object.

static SubstructureSearch.**from_xml** (*xml*)
Create a substructure search from XML.

Parameters: **xml** -- XML string

static SubstructureSearch.**from_xml_file** (*file_name*)
Create a substructure search from an XML file.

Parameters: **file_name** -- path to XML file

SubstructureSearch.read_xml (*xml*)
Read search query from XML.

Parameters: **xml** -- XML string

SubstructureSearch.read_xml_file (*file_name*)
Read search parameters from an XML file.

Parameters: **file_name** -- path to XML file
Raises: IOError if the file cannot be read

class ccdc.search.**ReducedCellSearch** (*query=None*, *settings=None*)
Provide reduced cell searches.

class **CrystalQuery** (*crystal*)
Reduced cell query from a crystal.

class ReducedCellSearch.**Query** (*lengths=None*, *angles=None*, *lattice_centring=None*)
Base query.

class ReducedCellSearch.**Settings** (*_settings=None*)
Settings appropriate to a reduced cell search.

absolute_angle_tolerance
The absolute angle tolerance.

is_normalised
Whether the input cell is normalised.

percent_length_tolerance
The cell length tolerance as a percentage of the longest cell dimension.

reset ()
Reset to default values.

class ReducedCellSearch.**XMLFileQuery** (*fname*)
Reduced cell query from a file name.

class ReducedCellSearch.**XMLQuery** (*xml*)
Reduced cell query from an XML representation.

ReducedCellSearch.compare_cells (*r0*, *r1*)
Compare two reduced cells.

Parameters:
- **r0** -- the first reduced cell, an instance of **ccdc.crystal.Crystal.ReducedCell**
- **r1** -- the second reduced cell similarly

Returns: boolean

static ReducedCellSearch.**from_xml** (*xml*)
Construct a reduced cell search from an XML representation.

Parameters: **xml** -- XML string

static ReducedCellSearch.**from_xml_file** (*file_name*)
Construct a reduced cell search from an XML file.

Parameters: **file_name** -- path to XML file

ReducedCellSearch.read_xml (*xml*)
Read XML into this ReducedCellSearch.

Parameters: **xml** -- XML string

ReducedCellSearch.read_xml_file (*file_name*)
Read an XML file into this ReducedCellSearch.

Parameters: **file_name** -- path to XML file

Raises: IOError if the file cannot be read

ReducedCellSearch.set_query (*query*)
Set the query.

Conformer API

Introduction

The **ccdc.conformer** module contains classes concerned with molecular conformations.

The three main classes of the **ccdc.conformer** module are:

- **ccdc.conformer.MoleculeMinimiser**
- **ccdc.conformer.ConformerGenerator**
- **ccdc.conformer.GeometryAnalyser**

A **ccdc.conformer.MoleculeMinimiser** can be used to optimise the bond distances and valence angles of a 3D input molecule using the **ccdc.conformer.MoleculeMinimiser.minimise()** function:

```
from ccdc.conformer import MoleculeMinimiser
molecule_minimiser = MoleculeMinimiser()
minimised_mol = MoleculeMinimiser.minimise(mol)
```

A **ccdc.conformer.ConformerGenerator** can be used to generate a set of conformers for an input molecule using the **ccdc.conformer.ConformerGenerator.generate()** function:

```
from ccdc.conformer import ConformerGenerator
from ccdc.io import MoleculeWriter
conformer_generator = ConformerGenerator()
conformers = conformer_generator.generate(mol)
with MoleculeWriter('conformers.mol2') as mol_writer:
    for c in conformers:
        mol_writer.write(c.molecule)
```

A **ccdc.conformer.GeometryAnalyser** can be used to analyse the geometry of an input molecule using a knowledge-based library of intramolecular geometries based on the CSD.

The **ccdc.conformer.GeometryAnalyser** class contains nested classes:

- **ccdc.conformer.GeometryAnalyser.Settings**
- **ccdc.conformer.GeometryAnalyser.Analysis**
- **ccdc.conformer.GeometryAnalyser.AnalysisHit**

The **ccdc.conformer.GeometryAnalyser.analyse_molecule()** function can be used to validate the complete geometry of a given query structure.

```
>>> from ccdc.io import EntryReader
>>> csd_reader = EntryReader('CSD')
>>> yigpio01 = csd_reader.molecule('YIGPIO01')
```

```
>>> from ccdc.conformer import GeometryAnalyser
>>> analysis_engine = GeometryAnalyser()
>>> checked_mol = analysis_engine.analyse_molecule(yigpio01)
>>> for tor in checked_mol.analysed_torsions:
...     if tor.unusual:
...         print('%s %d %.2f' % (str(tor.atom_labels), tor.nhits, tor.local_density))
...
[u'O4', u'C31', u'N5', u'C24'] 2234 3.45
[u'O5', u'C31', u'N5', u'C24'] 2230 3.86
[u'O5', u'C32', u'C33', u'S1'] 76 2.63
```

Seealso

Descriptive conformer generation and molecular minimisation documentation and the *descriptive molecular geometry analysis documentation.*

API

Knowledge base version number

ccdc.conformer.**_mogul_version** ()
The version of mogul being used.

Molecule minimisation

class ccdc.conformer.**MoleculeMinimiser** (*nthreads=1*)
Minimises a single or a list of molecules.

minimise (*mol*)
Return a minimised copy of the input molecule.
This makes use of the Tripos force field functional forms.
However, where available equilibrium bond distances and valence angles are parameterised using data obtained from CSD distributions.

Parameters: **mol** -- **ccdc.molecule.Molecule**
Returns: **ccdc.molecule.Molecule**

Conformer generation

> ***Note***
>
> The **`ConformerGenerator`** class is available only to CSD-Discovery, CSD-Materials and CSD-Enterprise users.

class ccdc.conformer.**ConformerGenerator** (*settings=None*, *skip_minimisation=False*, *nthreads=1*)

Generates conformers for a single or a list of molecules.
This functionality is available only under licenced conditions. Please contact support@ccdc.cam.ac.uk for details.

generate (*mols*)

Generate conformers for supplied molecule(s).

Parameters: **mols** -- a **`ccdc.molecule.Molecule`** or a list of **`ccdc.molecule.Molecule`**

Returns: a **`ccdc.conformer.ConformerHitList`** or a list of **`ccdc.conformer.ConformerHitList`** instances

Note that conformers cannot be generated for molecules with missing coordinates. These will be ignored if they occur in the input.

class ccdc.conformer.**ConformerHitList** (*identifier*, *_dr*)

A conformer generator result.

distributions_pruned

Whether or not the geometry distributions were pruned in order to perform an exhaustive search.

flexible_rings

The flexible rings considered by the generator.

max_log_probability

Maximum log probability.

min_log_probability

Minimum log probability.

minimised_molecule

The minimised molecule from which conformers were generated.

n_flexible_rings_in_molecule

The number of flexible rings in the molecule.

n_flexible_rings_sampled

The number of flexible rings sampled by the generator.
This may be smaller than the number of rings in the input molecule if there are no data in the CSD for the ring.

n_flexible_rings_with_no_observations

Number of flexible rings for which no crystallographic data is available.

n_matched_rotamers

Rotamers which have been matched in the fragment_library.txt parameter file.

n_rotamers_in_molecule

The number of rotamers in the molecule.

n_rotamers_sampled

The number of rotamers sampled by the generator.
This may be smaller than the number of rotamers in the input molecule if there are no data in the CSD for the rotamer.

n_rotamers_with_no_observations
Number of rotamers for which no crystallographic data is available.

original_molecule
The input molecule.

rotamers
The rotamers considered by the generator.

rotamers_with_no_observations
The list of bonds for which the CSD was unable to provide enough input data.

sampling_limit_reached
Whether the internal sampling limit as been reached.

class ccdc.conformer.**ConformerHit** (*mol*, *parent*)
An individual conformer.

normalised_score
Normalised score associated with this conformer (0 = best, 1 = worst).

probability
Probability associated with this conformer.

rmsd (*wrt='original'*, *exclude_hydrogens=True*)
Return the RMSD of this conformer with respect to the original or the minimised molecule.

Parameters:
- **wrt** -- either 'original' or 'minimised'
- **exclude_hydrogens** -- boolean

Returns: float

class ccdc.conformer.**ConformerSettings**
Settings for conformer generation.
Any settings that are set to None will be set to the system defaults.

max_conformers = *None*
Maximum number of conformers to generate.

max_unusual_torsions = *None*
Number of unusual torsions allowed per confomer.

normalised_score_threshold = *None*
Maximum deviation from the theoretically achievable conformer probability normalised (0="best", 1="worst").

superimpose_conformers_onto_reference = *None*
Whether or not to superimpose to a common reference.

Geometry analysis

class ccdc.conformer.**GeometryAnalyser** (*settings=None*, *databases=None*)
The geometry analysis engine.

class **Analysis** (*analysis*, *mol*, *classification*, *settings*, *siteless*)
A single geometric analysis for a specific bond, angle, torsion or ring feature.

atom_labels
The labels of atoms in the reference fragment.

d_min

Return the distance to the nearest observed value.
If rawscore is not specified, the geometric value of the query fragment will be used.

distribution
List of numeric values found by the search.

enough_hits
Whether there be enough hits for a sound judgement.

few_hits
Whether there be enough hits for a sound judgement.

fragment_label
Underscore separated string of atom labels.

generalised
Whether or not the analysis for this fragment resulted from a generalised search.

histogram (*bin_size=None*, *minimum=None*, *maximum=None*)
Return the histogram of the distribution as a tuple of integers.
This function puts the distribution values into bins according to the criteria specified.

Parameters:
- **bin_size** -- defaults to (maximum - minimum)/40 if set to None
- **minimum** -- defaults to smallest observed value if set to None
- **maximum** -- defaults to largest observed value if set to None

Returns: tuple of integers

hit_identifiers
List of molecule identifiers of the hits in the distribution.

hit_molecules
The list of molecules hit by this result.

hits
List of **ccdc.conformer.GeometryAnalyser.AnalysisHit** instances found by the search.
Note that the features below can be extracted from an **ccdc.conformer.GeometryAnalyser.AnalysisHit**:

- **ccdc.conformer.GeometryAnalyser.AnalysisHit.molecule**
- **ccdc.conformer.GeometryAnalyser.AnalysisHit.atom_indices**
- **ccdc.conformer.GeometryAnalyser.AnalysisHit.atom_labels**
- `value` of the geometric feature in the hit

For more information see the **ccdc.conformer.GeometryAnalyser.AnalysisHit** documentation.

local_density
Local density of the distribution around the query value.

lower_quartile
The lower quartile of the distribution.

maximum
The maximum of the distribution.

mean
The mean of the distribution.

median
The median of the distribution.

minimum
The minimum of the distribution.

nhits
The number of hits in the distribution.

no_hits
Whether the fragment has no data within the CSD.

percentile (*p*)
Return the percentile of the observed value.
Note that this function will raise a TypeError if the value (p) is not in between 0 and 1.

standard_deviation
The standard deviation of the distribution.

type
The type of geometric feature represented by this result.
In other words was this **ccdc.conformer.GeometryAnalyser.Analysis** derived from a bond, angle, torsion or ring analysis.

unusual
Check if the geometric feature is unusual or not.
If the enough_hits and few_hits parameters are set to True (default behaviour) this function will return True if the geometric feature is classified as unusual.
If the few_hits parameter is set to False this function will only return True if the geometric feature is unusual and there are enough hits to support this claim.
If the enough_hits parameter is set to False this function will only return True if the geometric feature is unusual and there is not enough hits to support this claim.
If both the enough_hits and few_hits parameter are set to False then this function will always return False.

upper_quartile
The upper quartile of the distribution.

value
Geometric value represented by the reference fragment.

z_score
Return the zscore of the observed value.

class GeometryAnalyser.**AnalysisHit** (*refcode*, *source*, *value*, *_analysis*, *_distrib*, *_index*)
A single geometry analysis hit fragment.
In other words one of the observations that make up the geometry analysis distribution.

atom_indices
The indices of the matched atoms in the hit molecule.

atom_labels
The labels of the matched atoms in the hit molecule.

atoms
The atoms of a hit.

bond_length
The bond length of the hit fragment.

Raises: TypeError if the hit is not for a bond length

crystal
The hit crystal.

entry
The hit entry.

identifier
The identifier of the hit.

molecule
The hit molecule.

source_name
The name of the source of the hit.

torsion_angle
The torsion angle of the hit fragment.

Raises: TypeError if the hit is not for a torsion angle

valence_angle
The valence angle of the hit fragment.

Raises: TypeError if the hit is not for a valence angle

class GeometryAnalyser.**Settings**
Controls the operation of the geometry analyser.

class **GeometrySettings** (*identifier*, *type*, *settings*)
Settings for a particular fragment type.
In other words settings that are applied to one of the below:

- Bond distances
- Valence angles
- Torsion angles
- Ring RMSDs

analyse
Whether to analyse this fragment type.

classification_measure
How to measure whether an observation is unusual.

classification_measure_threshold
The value at which an observation will be found to be unusual.

few_hits_threshold
Threshold below which a distribution is considered to have too few hits.

local_density_threshold
Local density threshold used to classify torsions and rings as unusual.
Note that the local density is irrelevant for bonds and angles.

local_density_tolerance
The local density tolerance.

min_obs_exact
Minimum acceptable size of an exact distribution.
If there is no distribution containing at least this number of observations the geometry analyser will perform a generalised search according to the criteria specified by other settings.

min_obs_generalised
Minimum number of observations that the geometry analyser should try to find.

If this is 0 then generalised searches will never be performed.
Similarly, if generalisation has been turned off this setting will not have an effect.

min_relevance
Relevance criterion for a generalised hit to be accepted.
The geometry analyser determines how similar a fragment is to the query by calculating a relevance value. The min_relevance setting tells the geometry analyser to accept, in a generalised search, only fragments whose relevance is equal to or greater than this threshold.

summary ()
Return a summary the settings as a string.

zscore_threshold
Z-score threshold used to classify bonds and angles as unusal.
Note that the z-score is irrelevant for torsions and rings.

GeometryAnalyser.Settings.**generalisation**
Setting determining if searches should be generalised or not.

GeometryAnalyser.Settings.**heaviest_element**
Filter on heaviest element.
This setting tells the geometry analyser to ignore hits from CSD structures that have elements heavier than that for a specified atomic symbol.
The atomic symbol is case sensitive.

GeometryAnalyser.Settings.**impose_upper_limits**
Whether there an upper limit imposed on generalised searches or not.
This setting tells the geometry analyser whether or not to limit the number of levels traversed for generalised searches. Occasionally the geometry analyser can take a very long time to identify similar fragments when performing a generalised search. Limiting the number of levels traversed will reduce the chances of this happening but may also result n fewer hits being found.

GeometryAnalyser.Settings.**organometallic_filter**
Configure how organometallic and organic hits should be filtered.
This setting instructs the geometry analyser to ignore fragments depending on whether they are from organic or organometallic structures.
There are three possible options for this setting:

- 'all'
- 'metalorganics_only'
- 'organics_only'

GeometryAnalyser.Settings.**rfactor_filter**
Filter on R-factor.
Note that there are only four possible settings for this option:

- 0.05
- 0.075
- 0.1
- any

However you can set the filter using any value and the appropriate filter will be selected. Note that if the value supplied is greater than 0.1 this means that the R-factor filter will be set to None. If you set the filter to None or 'any' the filter will also be set to None.

GeometryAnalyser.Settings.**solvent_filter**
Configure how solvents and non-solvents should be filtered.
This setting instructs the geometry analyser to ignore fragments depending on whether they are from solvent or non-solvent molecules.
There are three possible options for this setting:

- 'include_solvent'
- 'exclude_solvent'
- 'only_solvent'

GeometryAnalyser.Settings.summary ()
Return a summary the settings as a string.

GeometryAnalyser.analyse_angle (*a*, *b*, *c*)
Perform a geometry analysis on a single valence angle.

Params a: **ccdc.molecule.Atom**
Params b: **ccdc.molecule.Atom**
Params c: **ccdc.molecule.Atom**
Returns: **ccdc.conformer.GeometryAnalyser.Analysis**
Raises: TypeError raised if the atoms supplied do not make up a bonded angle

GeometryAnalyser.analyse_bond (*a*, *b*)
Perform a geometry analysis on a single bond.

Params a: **ccdc.molecule.Atom**
Params b: **ccdc.molecule.Atom**
Returns: **ccdc.conformer.GeometryAnalyser.Analysis**
Raises: TypeError raised if the atoms supplied do not form a covalent bond

GeometryAnalyser.analyse_molecule (*mol*)
Perform a geometry analysis of the whole molecule.

Params mol: **ccdc.molecule.Molecule** to be analysed
Returns: **ccdc.molecule.Molecule** augmented with analysis data

GeometryAnalyser.analyse_ring (ats*)**
Perform a geometry analysis on a single ring.

Params ats*: **ccdc.molecule.Atom** instances that make up the ring
Returns: **ccdc.conformer.GeometryAnalyser.Analysis**
Raises: TypeError raised if the atoms supplied do not make up a ring

GeometryAnalyser.analyse_torsion (*a*, *b*, *c*, *d*)
Perform a geometry analysis on a single torsion angle.

Params a: **ccdc.molecule.Atom**
Params b: **ccdc.molecule.Atom**
Params c: **ccdc.molecule.Atom**
Params d: **ccdc.molecule.Atom**
Returns: **ccdc.conformer.GeometryAnalyser.Analysis**
Raises: TypeError raised if the atoms supplied do not make up a bonded torsion

GeometryAnalyser.fragment_identifier (*fragment*)
The unique identifier of a particular type of fragment.
This is a string encoding the molecular environment of a fragment.

Parameters: **fragment** -- an instance of **ccdc.conformer.GeometryAnalyser.Analysis**
Returns: a string of four numbers separated by colons.

Protein API

Introduction

protein.py - representation of a protein.

James' thoughts:

- To remove residues, could you do something like del protein['A']['ASN27']
- Could chains return an ordereddict?
- Could chains and residues subclass Molecule? (are they molecules?)
- Move cavity stuff to docking results?
- Replace file_name constructor parameter with from_string.

API

class ccdc.protein.**Protein** (*identifier*, *_molecule=None*)

class **Chain** (*index*, *_protein_structure=None*)
A chain of a protein.

residues
The residues of a chain.

sequence
The sequence of amino acid one letter codes in this chain.

class Protein.**Residue** (*_residue*)
A single amino acid residue of a protein.

atoms
The atoms of the residue.

backbone_atoms
The backbone atoms of the amino acid.

c_alpha
The C alpha atom of the residue.

c_beta
The C beta atom, or None if there is no C beta atom.

c_terminus
The C terminus atom.

carbonyl_oxygen
The carbonyl oxygen atom.

chain_identifier
The identifier of the chain of which this residue is a part.

cysteine_sulphur
The sulphur of a cysteine residue, or None if not a cysteine.

identifier
The identifier of this residue.

is_acidic
Whether the residue is acidic.

is_basic
Whether the residue is basic.

is_hydrophilic
Whether the residue is hydrophilic.

is_hydrophobic
Whether the residue is hydrophobic.

n_terminus
The N terminus atom.

one_letter_code
The one letter code of the amino acid.

sidechain_atoms
The sidechain atoms of this amino acid.

three_letter_code
The three letter code of the amino acid.

Protein.**cavity_atoms**
The atoms making up the binding site, if this was read from a gold protein.

Protein.**cavity_residues**
The residues making up the cavity.

Protein.**chains**
A tuple of **ccdc.protein.Protein.Chain**.

static Protein.**from_entry** (*entry*)
Constructs a protein fram a **ccdc.entry.Entry**.

static Protein.**from_file_name** (*file_name*)
Reads a protein from a file, and constructs the protein.

Protein.**ligands**
The tuple of ligands in the protein.
The identifier of the molecule is of the form chain_id:residue_name.

Protein.**metals**
The metal atoms of the protein.

Protein.remove_all_waters ()
Removes all waters from the protein.

Protein.remove_chain (*chain_id*)
Remove the chain with the given identifier.

Protein.remove_ligand (*ligand_id*)
Remove the specified ligand.

Parameters: **ligand_id** -- str, of the form chain_id:ligand_id.

Protein.remove_metal (*atom*)
Remove the given metal atom.

Protein.remove_residue (*residue_id*)
Remove the specified residue.

Protein.remove_water (*atom*)
Remove the water with the given oxygen atom.

`Protein.`**`residues`**
The amino acid residues of the protein.

`Protein.`**`sequence`**
The one-letter code sequence.

`Protein.`**`waters`**
The waters of the protein.

Returns: a tuple of **`ccdc.molecule.Atom`**, representing the oxygens of the water.

Docking API

Introduction

> ### *Note*
>
> The **`ccdc.docking`** module is under development - currently available only to associated collaborators.

docking.py - API for docking

API

class `ccdc.docking.`**`Docker`** (*`settings=None`*)
Docker.

class **`Results`** (*`settings`*, *`return_code=None`*, *`pid=None`*)
Docking results.

`docking_log`
The content of the docking log file.

`error_log`
The content of the docking error log file.

`ligand_log (`*`index`*`)`
The content of a ligand log.

`ligands`
The ligands of the docking.
The value of this property is a **`ccdc.io.EntryReader`**. Each entry has an attributes property, a dictionary of the docking information pertaining to the docking.

`make_complex (`*`ligand`*`)`
Make the complex with the ligand, adjusting rotatables as required.

Returns: a **`ccdc.protein.Protein`** with the ligand added.

`protein_log`
The content of the protein log file.

`proteins`
The protein(s) of the docking.

Returns: a tuple of **`ccdc.protein.Protein`**.

This tuple will have more than one entry if ensemble docking was used.

class Docker.**Settings** (*_settings=None*)
Settings for docker.

class **BindingSite** (*origin=(0*, *0*, *0)*, *radius=12.0*)
Information about a docking binding site.

class Docker.Settings.**LigandFileInfo** (*file_name*, *ndocks=1*, *start=0*, *finish=0*)
Information about a ligand file.

class Docker.Settings.**ProteinFileInfo** (*file_name=None*, *_protein_data=None*)
Data associated with a protein for docking.

file_name
The file name of the protein.

Docker.Settings.add_ligand_file (*file_name*, *ndocks=1*, *start=0*, *finish=0*)
Add a file of ligands to the docking settings.

Parameters:
- **file_name** -- a mol2 or sdf file of ligand molecules, or a **ccdc.docking.Docker.Settings.LigandFileInfo** instance.
- **ndocks** -- int, the number of docking attempts for each ligand
- **start** -- int, index of ligand at which to start
- **finish** -- int, index of ligand at which to finish

Docker.Settings.add_protein_file (*file_name*)
Add a target file to be docked against.

Docker.Settings.add_target (*molecule*)
Add a target molecule.

Docker.Settings.**autoscale**
The autoscale percentage, which controls how much searching is performed.
The docker will determine how much docking is reasonable to perform on a ligand based on the number of rotatable bonds and the number of hydrogen donors and acceptors. This percentage will scale the amount of docking done to perform faster or more thorough docking.

Docker.Settings.**binding_site**
The binding site for the docking.

Docker.Settings.clear_ligand_files ()
Remove all ligand datafiles from settings.

Docker.Settings.clear_protein_files ()
Clear the set of targets.

Docker.Settings.**diverse_solutions**
Diverse solutions settings.
If diverse solutions is enabled this will be (True, cluster size, rmsd), otherwise (False, None, None)

Docker.Settings.**early_termination**
Whether early termination is permitted.
If early termination is permitted this will be (True, number_of_solutions, rmsd_threshold), if not this will be (False, None, None)

Docker.Settings.**fitness_function**
Which fitness function to use.
Options are 'goldscore', 'chemscore', 'asp', 'plp', 'consensus_score'

static Docker.Settings.**from_file** (*file_name*)

Read docking settings from a gold.conf file.

`Docker.Settings.`**`ligand_files`**
The ligand datafile settings.

Returns: tuple of class:*ccdc.docking.Docker.Settings.LigandFileInfo* instances.

`Docker.Settings.make_absolute_file_names (`*`file_name`*`)`
Convert any relative file names to absolute file names.

Parameters: **file_name** -- str, the location of the settings file.

`Docker.Settings.`**`output_directory`**
Directory to which output will be sent.

`Docker.Settings.`**`output_file`**
Output file.
If this is an empty string then each docking will be in a separate file.

`Docker.Settings.`**`output_format`**
Desired format for output file.

`Docker.Settings.`**`protein_files`**
The protein file targets.

`Docker.Settings.`**`rescore_function`**
The fitness function used for rescoring.
Should not be the same as the fitness function.

`Docker.Settings.set_hostname (`*`hostname='localhost'`*`, `*`ndocks=1`*`)`
Set the hostname on which docking jobs will be run.

`Docker.Settings.write (`*`file_name`*`)`
Write the settings to a file.

Parameters: **file_name** -- str, name of the file to which to write

`Docker.copy_settings (`*`newdocker`*`)`
Copy this docker's settings to another docker instance

`Docker.dock (`*`file_name=None`*`, `*`mode='foreground'`*`)`
Dock from the current settings.

Parameters:

- **file_name** -- file name of the settings. If None, the current settings will be written to a temporary directory.
- **mode** -- one of 'foreground', 'background' or 'interactive'.

`Docker.dock_status ()`
Check the status of a docking job via the gold.pid file.

`Docker.`**`results`**
The docking results.
If the docking is still in progress, the results may be partial.

Screening API

Introduction

> ***Note***
>
> The **`ccdc.screening`** module is under development - currently available only to associated collaborators.

The **`ccdc.screening`** module can be used to screen a library of compounds against a pharmacophore query obtained from one or multiple overlaid ligands. The algorithm generalises the 3D pharmacophore definition using atom property fields that are created around the query based on user-defined atom types and potentials.

> **Seealso**
>
> *Descriptive field-based virtual screening documentation*

API

Ligand Screener

class ccdc.screening.**Screener** (*overlay*, *settings=None*, *nthreads=1*)
Performs field-based ligand screening.

class **ScreenHitList** (*confs*, *atom_types*, *_drs*)
List of screening results.

class **ScreenHit** (*molecules*, *atom_types*, *_dr*)
An individual screening hit.

score
The screening score. A lower score is better.

Screener.ScreenHitList.**best_hit**
The hit with the lowest score.

class Screener.**Settings**
Screener settings.

bias_conformer_selection
Whether or not to bias the conformer selection to low-energy conformers.
This assumes that lower-energy conformations come earlier in the list, such as for the CSD conformer generator.

excluded_volume_envelop
Size of the excluded volume envelope.

excluded_volume_penalty
Penalty to be applied for atoms in the excluded volume.

fitting_points_cluster_radius
If the distance between two fitting points is less than the cluster radius, the fitting point with the higher score will be eliminated.

fitting_points_threshold
Grid points with a score lower than this this threshold will create a fitting point.

output_directory
Location where data files may be stored.

parameter_directory
Location of parameter files.

save_files
Whether or not to save files.

store_atom_types
Whether or not to store atom types.

Screener.screen (*molecules*)
Screen conformers against the overlay.

Parameters: **molecules** -- a list of lists of **ccdc.molecule.Molecule**
Returns: **ccdc_internal.Screener.ScreenHitList**

Interaction API

Introduction

InteractionLibrary is a knowledge-based library of intermolecular interactions.

The **ccdc.interaction** module gives access to the InteractionLibrary central and contact groups. It also provides an interface for working with the data stored in .istr files.

```
>>> from ccdc.interaction import InteractionLibrary
>>> central_lib = InteractionLibrary.CentralGroupLibrary()
>>> contact_lib = InteractionLibrary.ContactGroupLibrary()
>>> central_ketone = central_lib.group_by_name('aliphatic-aliphatic ketone')
>>> contact_alcohol = contact_lib.group_by_name('alcohol OH')
>>> interaction_data = central_ketone.interaction_data(contact_alcohol)
>>> print 'Rel. density %.2f, est. std.dev. %.2f' % interaction_data.relative_density
Rel. density 2.96, est. std.dev. 0.28
```

Methods to establish which central and contact groups are present in a molecule are provided.

```
>>> from ccdc.io import MoleculeReader
>>> csd_molecule_reader = MoleculeReader('CSD')
>>> aabhtz = csd_molecule_reader.molecule('AABHTZ')
>>> aabhtz_central_group_hits = central_lib.search_molecule(aabhtz)
>>> len(aabhtz_central_group_hits)
8
```

The order of the central group hits is indeterminate. However, we do know that methyl is present in AABHTZ.

```
>>> names = [g.name for g in aabhtz_central_group_hits]
>>> assert('methyl' in names)
>>> i = names.index('methyl')
>>> print aabhtz_central_group_hits[i].match_atoms()
[Atom(C10), Atom(C11), Atom(H7), Atom(H8), Atom(H9)]
```

Seealso

Descriptive Interaction Library documentation.

API

class ccdc.interaction.**InteractionLibrary**
The InteractionLibrary derived from CSD data.

class **CentralGroup** (*_group*)
central group definition.

contact_groups ()
Return list of contact groups for which the interaction library has data.
The **ccdc.interaction.InteractionLibrary.ContactGroup** instances returned are the ones for which the specific **ccdc.interaction.InteractionLibrary.CentralGroup** has data.

Returns: list of **ccdc.interaction.InteractionLibrary.ContactGroup** instances

interaction_data (*contact_group*)
Return the file for this central group and the contact group.

Parameters: **contact_group** -- **ccdc.interaction.InteractionLibrary.ContactGroup**
Returns: **ccdc.interaction.InteractionLibrary.InteractionData**

class InteractionLibrary.**CentralGroupLibrary**
Collection of central groups.
groups list of **ccdc.interaction.InteractionLibrary.CentralGroup** instances in the library

class InteractionLibrary.**ContactGroup** (*_group*)
contact group definition

central_groups ()
List of central groups which have data for this contact group

Returns: list of **ccdc.interaction.InteractionLibrary.CentralGroup** instances

interaction_data (*central*)
The **ccdc.interaction.InteractionLibrary.InteractionData** corresponding to the given central group and this contact group

Parameters: **central** -- **ccdc.interaction.InteractionLibrary.CentralGroup**

class InteractionLibrary.**ContactGroupLibrary**
Collection of contact groups.
groups list of **ccdc.interaction.InteractionLibrary.ContactGroup** instances in the library

group_by_index (*index*)
Find a contact group by index.

Parameters: **index** -- integer index of the group to return
Returns: the **ccdc.interaction.InteractionLibrary.ContactGroup**

class InteractionLibrary.**FunctionalGroupHit** (*group*, *hit*)
A hit from a functional group search.

match_atoms (*indices=False*)
Delegated to **ccdc.search.SubstructureHit.match_atoms()**.

name
The name of the group matched the hit.

class InteractionLibrary.**InteractionData** (*fname*)
Class for working with interaction data.

average_density
Average density of the data.

central_group_name
Name of the central group.

contact_group_name
Name of the contact group.

histogram
Histogram of contact distances.

hits
The entry hits in this file.

max_distance
Furthest distance in the histogram.

min_distance
Closest distance in the histogram.

ncentral_atoms
Number of atoms in the central group.

ncontact_atoms
Number of atoms in the contact group.

ncontacts
Number of observations in the istr file.

ncontacts_in_range (*low*, *high*)
The number of observations in a given distance range

nflexible_atoms
Number of flexible atoms.

nvdw_contacts
Number of observations with a distance <= sum of VdW radii.

relative_density
Relative density and estimated standard deviation tuple.

class InteractionLibrary.**InteractionHit** (*_link*)
A hit from InteractionData

central_group_atoms (*indices=False*)
The atoms of the central group

contact_group_atoms (*indices=False*)
The atoms of the contact group

crystal
The crystal of the hit

distance
Distance between central and contact groups relative to Sum VdW

entry
The entry for the hit

identifier
The refcode of the hit

molecule

The molecule of the hit

Descriptors API

Introduction

The **ccdc.descriptors** module contains classes for calculating descriptors.

The main classes in the **ccdc.descriptors** module are:

- **ccdc.descriptors.MolecularDescriptors**.
- **ccdc.descriptors.GeometricDescriptors**.
- **ccdc.descriptors.PowderPattern**.

Seealso

Descriptive documentation of the descriptors module

Seealso

Descriptive documentation for geometric, molecular and crystallographic descriptors.

API

class ccdc.descriptors.**MolecularDescriptors**
Namespace for descriptors of a molecular nature.

class **MaximumCommonSubstructure** (*settings=None*)
Identifies the maximum common substructure of two molecules.

class **Settings**
Settings for the MCS calculation.

check_bond_count
Whether the bond count of an atom be checked.

check_bond_polymeric
Check whether the bond be polymeric.

check_bond_type
Whether the bond type be checked.

check_charge
Whether the atom charge be checked.

check_element
Whether the element be checked.

check_hydrogen_count
Whether the atom's hydrogen count be checked.

connected
Whether substructure should be connected.
Note that finding disconnected maximal substructures is a lot slower than finding connected.

ignore_hydrogens
Whether the hydrogens be ignored.

MolecularDescriptors.MaximumCommonSubstructure.search (*mol1*, *mol2*, *only_edges=False*)
Calculate the maximum common substructure between two molecules.

Parameters:
- **mol2** (*mol1,*) -- **ccdc.molecule.Molecule** instances.
- **only_edges** -- bool. The search will find a maximal common substructure matching only the edges.

Returns: a pair of tuples, giving matched **ccdc.molecule.Atom** and **ccdc.molecule.Bond** instances.

Note: this function is computationally exponential, so will take a long time on large molecules.

static MolecularDescriptors.**atom_angle** (*a*, *b*, *c*)
Angle subtended by three arbitrary atoms.

Parameters:
- **a** -- **ccdc.molecule.Atom**
- **b** -- **ccdc.molecule.Atom**
- **c** -- **ccdc.molecule.Atom**

Returns: float - the angle in degrees or None if one of the atoms has no coordinates

static MolecularDescriptors.**atom_centroid** (**atoms*)
Define the centroid of the given atoms.

static MolecularDescriptors.**atom_distance** (*a*, *b*)
Distance between two atom irrespective their parent molecules.

Parameters:
- **a** -- **ccdc.molecule.Atom**
- **b** -- **ccdc.molecule.Atom**

Returns: float or None if one of the atoms has no coordinates

static MolecularDescriptors.**atom_plane** (**atoms*)
Define a plane from the coordinates of the atoms.

Parameters: **atoms** -- there must be at least three **ccdc.molecule.Atom** in the arguments.

static MolecularDescriptors.**atom_torsion_angle** (*a*, *b*, *c*, *d*)
Plane angle subtended by the triples abc and bcd.

Parameters:
- **a** -- **ccdc.molecule.Atom**
- **b** -- **ccdc.molecule.Atom**
- **c** -- **ccdc.molecule.Atom**
- **d** -- **ccdc.molecule.Atom**

Returns: float - the angle in degrees or None if one of the atoms has no coordinates

static MolecularDescriptors.**atom_vector** (*atom0*, *atom1*)
Define the vector from atom0 to atom1.

static MolecularDescriptors.**bond_length** (*bond*)
The length of a bond.

Parameters: **bond** -- **ccdc.molecule.Bond**

Returns: float, or None if an atom of the bond has no coordinates

static MolecularDescriptors.**overlay** (*mol1*, *mol2*, *atoms=None*, *invert=False*, *rotate_torsions=False*, *with_symmetry=True*)
Overlay mol2 on mol1.

Parameters:
- **mol1** -- a **ccdc.molecule.Molecule** instance
- **mol2** -- a **ccdc.molecule.Molecule** instance
- **atoms** -- a list of pairs of atoms to use in the overlay, or None for all atoms to be used
- **invert** -- allow inversions in the overlay
- **rotate_torsions** -- allow torsional rotations when overlaying
- **with_symmetry** -- take account of symmetry when overlaying atoms

Returns: a **ccdc.molecule.Molecule** instance which is a copy of mol2 overlayed on mol1

static MolecularDescriptors.**point_group_analysis** (*mol*)
Return Schoenflies notation of the point group symmetry of a molecule.
The point group symmetry is returned as a tuple of:

- order (e.g. 1)
- symbol (e.g. 'C1')
- description (e.g. 'Objects in this point group have no symmetry.')

Parameters: **mol** -- **ccdc.molecule.Molecule**

Returns: (int, str, str)

static MolecularDescriptors.**ring_centroid** (*ring*)
The centroid of the ring's atoms.

Parameters: **ring** -- **ccdc.molecule.Molecule.Ring**

static MolecularDescriptors.**ring_plane** (*ring*)
The plane of the ring's atoms.

Parameters: **ring** -- **ccdc.molecule.Molecule.Ring**

static MolecularDescriptors.**rmsd** (*mol1*, *mol2*, *atoms=None*, *overlay=False*, *exclude_hydrogens=True*, *with_symmetry=True*)
Return the RMSD of two molecules.
Both molecules should have the same atoms if :param atoms: is None.

Parameters:
- **atoms** -- a list of pairs **ccdc.molecule.Atom** or None
- **overlay** -- whether or not to overlay the molecules before calculating an RMSD
- **exclude_hydrogens** -- whether an all-atom or heavy atom RMSD should be calculated
- **with_symmetry** -- whether to allow symmetrical matches

Returns: float

class ccdc.descriptors.**GeometricDescriptors**
A namespace to hold geometric classes and functions.

class **Plane** (*vector*, *distance*, *_plane=None*)
A plane in 3D.

distance
The distance from the origin of the plane.

static **from_points** (**points*)
Construct a RMS fitted plane from points.

normal
The normal to the plane.

plane_angle (*plane*)

The angle between the two planes.

plane_distance (*plane*)
The shortest distance of the plane to another.

point_distance (*point*)
The distance of the point to the plane.

vector_angle (*vector*)
The angle between the plane and the vector.

class GeometricDescriptors.**Vector** (*x*, *y*, *z*)
A 3D vector.

static **from_points** (*p0*, *p1*)
Construct the vector from p0 to p1.

Parameters: **p1** (*p0,*) -- **ccdc.molecule.Coordinates**

static GeometricDescriptors.**point_angle** (*p0*, *p1*, *p2*)
The angle between three points.

static GeometricDescriptors.**point_distance** (*p0*, *p1*)
The distance between two points.

static GeometricDescriptors.**point_torsion_angle** (*p0*, *p1*, *p2*, *p3*)
The torsion angle between four points.

class ccdc.descriptors.**PowderPattern** (*_pattern*, *_settings=None*, *_simulation=None*)
Represents a powder pattern.

The powder pattern class is available only to CSD-Materials and
CSD-Enterprise users.

class **Settings**
Settings which may be set for a Powder simulation.
Setting None for any of the attributes will result in a default value being used.

deuterium_is_hydrogen = *None*
Whether deuterium and hydrogen are indistinguishable.

full_width_at_half_maximum = *None*
Peak width at half height (0.1).

include_hydrogens = *None*
Whether to include hydrogens.

second_wavelength = *None*
Optional second wavelength.

two_theta_maximum = *None*
Maximum value of two_theta (50.0).

two_theta_minimum = *None*
Minimum value of two_theta (5.0).

two_theta_step = *None*
Step size (0.02).

wavelength = *None*
Wavelength for the simulation.

class PowderPattern.**TickMark** (*_tick*)

A tick mark in a simulated powder pattern.

is_systematically_absent
Whether this tick mark is systematically absent.

two_theta
Two theta value of this tick.

class PowderPattern.**Wavelength** (*wavelength=None*, *scale_factor=1.0*)
Represents a wavelength for powder studies.
Some standard wavelengths - these are floats, not **ccdc.descriptors.PowderPattern.Wavelength**

scale_factor
The scale factor of this Wavelength.

wavelength
The wavelength.

PowderPattern.**esd**
The array of esd values (Estimated Square Deviations).

static PowderPattern.**from_crystal** (*crystal*, *settings=None*)
Create a PowderPattern from a crystal.

Parameters:
- **crystal** -- **ccdc.crystal.Crystal**
- **settings** -- **ccdc.descriptors.PowderPattern.Settings**

static PowderPattern.**from_xye_file** (*file_name*)
Create a PowderPattern from an xye file.

Parameters: **file_name** -- path to xye file

PowderPattern.integral (*start=0.0*, *end=180.0*)
The area under the curve.

Parameters:
- **start** -- float
- **end** -- float

Returns: float

PowderPattern.**intensity**
The array of intensity values.

PowderPattern.similarity (*other*)
Measure of match between this pattern and another.

Parameters: **other** -- **ccdc.descriptors.PowderPattern**

Returns: float

PowderPattern.**tick_marks**
The array of tick marks if this is a simulated powder pattern.

Returns: list of **ccdc.descriptors.PowderPattern.TickMark** or None if this is not a simulated powder pattern.

PowderPattern.**two_theta**
The array of two_theta values.

PowderPattern.write_raw_file (*file_name*)
Write a Bruker .raw file.

Parameters: **file_name** -- output file name

PowderPattern.write_xye_file (*file_name*)
Write a .xye format file.

Parameters: **file_name** -- output file name

Diagram API

Introduction

The **ccdc.diagram** module has functionality for generating 2D diagrams.

The main class of the **ccdc.diagram** module is **ccdc.diagram.DiagramGenerator**.

```
>>> from ccdc.io import EntryReader
>>> from ccdc.diagram import DiagramGenerator
>>> entry_reader = EntryReader('CSD')
>>> mol = entry_reader.molecule('ABEBUF')
>>> diagram_generator = DiagramGenerator()
>>> img = diagram_generator.image(mol)  # img is a PIL image
```

Seealso

Descriptive diagram documentation.

API

class ccdc.diagram.**DiagramGenerator** (*settings=None*)
Diagram generator.

class **Settings** (*_settings=None*)
Settings base class for diagram generation.

detect_intra_hbonds = *False*
Whether to detect intra-molecular hbonds.

explicit_polar_hydrogens = *False*
Whether to use explicit polar hydrogens.

font_size
The image font size.

highlight_color = *'red'*
Color used for highlighting atoms and bonds.

image_height
The image height.

image_width
The image width.

line_width
The image line width.

overwrite_existing_image = *False*
Whether to override the existing image of an entry.

return_type

The image method return type.

shrink_symbols = *True*
Whether the symbols should be shrunk.

DiagramGenerator.image (*argument*, *highlight_atoms=None*)
Return an image or list of images according to arguments.

Parameters:
- **argument** -- a **ccdc.crystal.Crystal**, a **ccdc.molecule.Molecule**, a **ccdc.entry.Entry**, or a list of them.
- **highlight_atoms** -- list of **ccdc.molecule.Atom**, or a list of atom lists

Returns: a PIL image, or a list of PIL images. It may return None if the diagram generation failed.

Utilities API

Introduction

The **ccdc.utilities** module contains general purpose classes.

The main classes of the **ccdc.utilities** module are:

- **ccdc.utilities.Logger**
- **ccdc.utilities.FileLogger**

Seealso

Descriptive utilities documentation.

API

class ccdc.utilities.**Logger**
Handles CCDC log messages and Python API messages.

fatal (*msg*, *contact=True*)
Log a critical message and exit.

Parameters:
- **msg** -- str
- **contact** -- bool (whether or not to include CCDC contact details in the message)

ignore_line_numbers (*tf=True*)
Format line numbers or not for output lines.

Parameters: **tf** -- bool

classmethod **reset** (*klass*)
Reset the logger.

set_ccdc_log_level (*value*)
Set the log level of the CCDC logger.

Parameters: **value** -- int

set_ccdc_minimum_log_level (*value*)
Set the minimum log level of the CCDC logger.
Note that a minimum log level of 1 produces an enormous amount of output. A minimum log level of 3 is recommended.

Parameters: **value** -- int

set_log_level (*value*)
Set the log level of Python log messages.

Parameters: **value** -- Logger.LOGLEVEL

set_output_file (*file_name*)
Specify an output file to use, rather than the default stdout.

Parameters: **file_name** -- str

class ccdc.utilities.**FileLogger** (*fname*)
A context manager to set logger output to a file.
Use it like this:

```
with FileLogger('/tmp/ccdc.log') as log:
    ...
    log.info('Something happened')
    ...
```

The file will be closed on exit and the logger reset to stderr.

Deprecated API documentation

None

INDEX

B

C

D

E

F

G

H

M

N

Q

R

S

Z