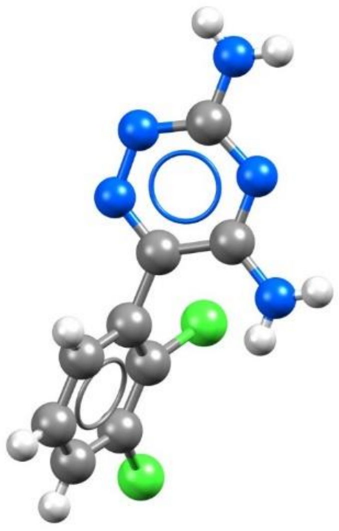
The CSD Python API: A Foundation for Innovation

Workshop Version 1.0 – July 2016
CSD V5.37



The Cambridge Crystallographic
Data Centre

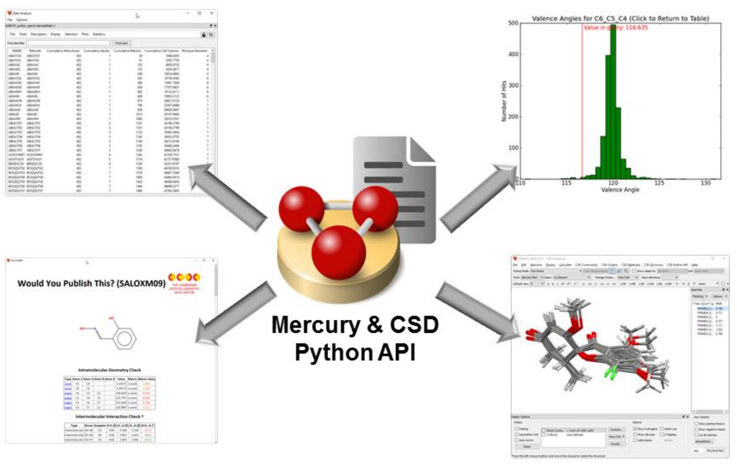
Introduction

Have you ever wanted just one extra little feature in Mercury, to just have one further parameter calculated from your structure or to be able to generate a slightly different format of output from CSD software? Well now you can add new functionality and output facilities directly to Mercury yourself, or persuade a friend to help you add it, via python scripting. The use of python scripts, a scripting language extremely popular now within the scientific community, from within Mercury opens up a lot of possibilities.

The CSD Python API provides access to the full breadth of functionality that is available within the various user interfaces (including Mercury, ConQuest, Mogul, IsoStar and WebCSD) as well as features that have never been exposed within an interface. Through python scripting it is possible to build highly tailored custom applications to help you answer detailed research questions, or to automate frequently performed analysis steps.

This tutorial will cover a range of aspects of the combined use of Mercury and the CSD Python API, building from an initial introduction to the mechanics of the CSD Python API menu in Mercury to advanced python scripting. The applications illustrated through these case studies are just as easily applied to your own experimental structures as they are to the examples shown here using entries in the Cambridge Structural Database (CSD).

The following exercises assume that you have a working knowledge of the program Mercury, in particular how to display and manipulate structures within the visualiser. There are many other features included with Mercury that are beyond the scope of this tutorial. For assistance with any aspect of Mercury, please ask your workshop instructor, or view the help guides included with the application.


Valence Angles for C6_C5_C4 (Click to Return to Table)
Value in query: 116.635
Number of Hits
Valence Angle
Would You Publish This? (SALOXM09)
Intramolecular Geometry Check
Intermolecular Interaction Check
Mercury & CSD Python API

CSD Python API scripts can be run from within Mercury to achieve a wide range of analyses, research applications and generation of automated reports

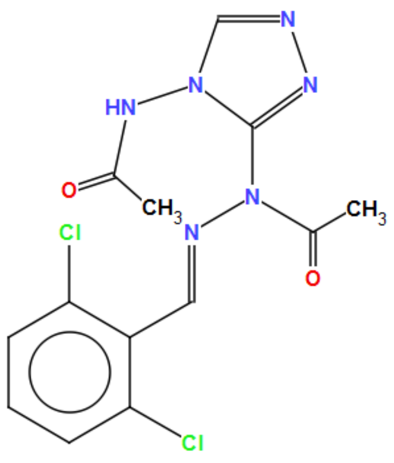
Case Study 1: Customising a simple script for use in Mercury

Aim

This case study will be focussing on the basics of how Mercury interacts with the CSD Python API, where scripts can be stored for use in Mercury and how to make small edits to an existing script. We will make use of a published crystal structure and a supplied python script, and then illustrate how to report some useful information about the structure that is not normally accessible from within Mercury.

Example system

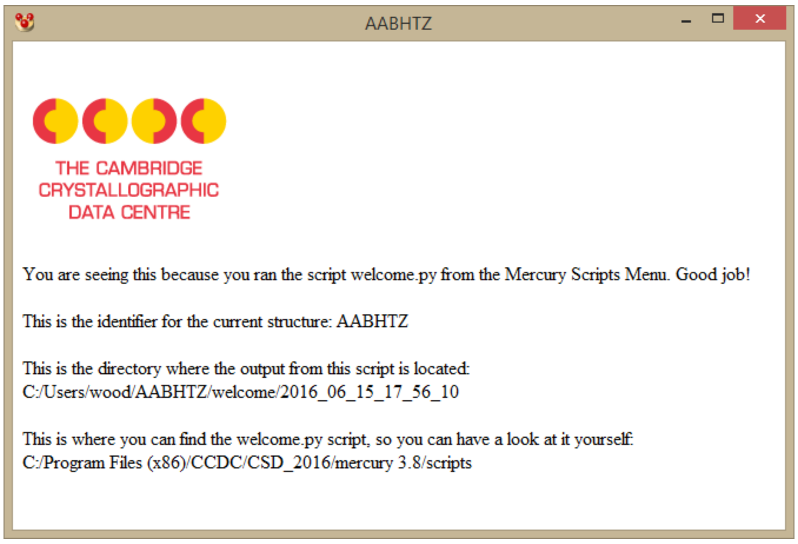
The example system we will be looking at for this case study is 4-acetoamido-3-(1-acetyl-2-(2,6-dichlorobenzylidene)hydrazine)-1,2,4-triazole (shown below) which happens to be the compound featured in the first entry of the Cambridge Structural Database (CSD) with the CSD refcode AABHTZ.



CSD refcode AABHTZ chemical diagram

Analysis

1. Launch Mercury by clicking its icon . In the Structure Navigator toolbar, type AABHTZ to bring up the first structure in the CSD.
2. From the top-level menu, choose **CSD Python API**, and then select **welcome.py** from the resulting drop-down menu. This will run a simple python script from within Mercury and illustrate the basics of how Mercury interacts with CSD Python API scripts.
3. Once the script has finished running, a new window will pop-up displaying the output of the script containing the CCDC logo and a few details about both the structure we are looking at and the set-up of your system.



You are seeing this because you ran the script welcome.py from the Mercury Scripts Menu. Good job!

This is the identifier for the current structure: AABHTZ

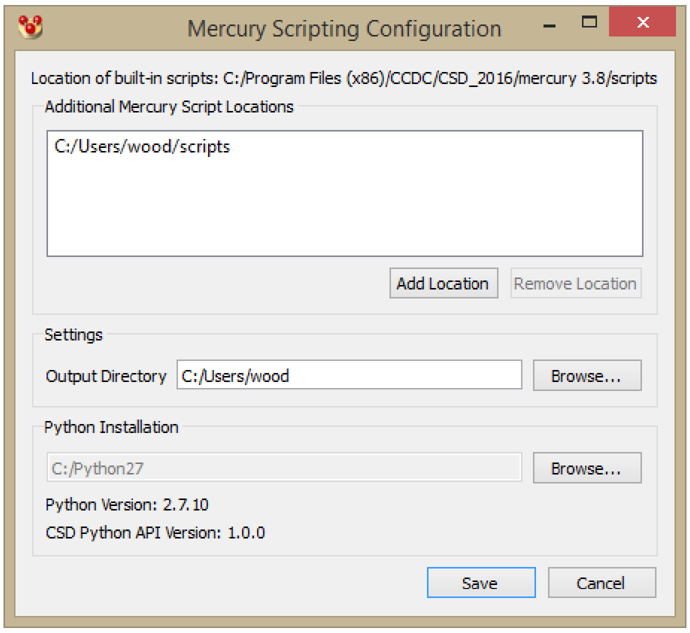
This is the directory where the output from this script is located:
C:/Users/wood/AABHTZ/welcome/2016_06_15_17_56_10

This is where you can find the welcome.py script, so you can have a look at it yourself:
C:/Program Files (x86)/CCDC/CSD_2016/mercury 3.8/scripts

4. The second line of text in the script output reports the identifier of the structure that we have displayed in the Mercury visualiser – AABHTZ – this is generated by the Python script and would change if we ran the script with another entry or other structural file displayed.
5. The third line of text in the script output reports exactly where the output file is located. The contents of this output window that popped up are encoded in a simple HTML file. Browse to the location shown using a file navigator on your computer (e.g. the File Explorer application on Windows). Right-click on the HTML file in that folder and open it with a file editor such as notepad – you should see that this file only contains a few lines of HTML text to produce the output you observed.

[For small scale editing of text files and Python scripts, such as the edits made in this case study, we would recommend Notepad++ (https://notepad-plus-plus.org/). For more in-depth Python editing or for interactive work, try looking at PyCharm (https://www.jetbrains.com/pycharm/) or iPython (https://ipython.org/).]

6. The fourth line of text in the script output reports where the actual script that you just ran is located – this will be contained within your Mercury installation directory. Browse to the folder location as before using a file navigator. This folder contains all the scripts bundled with the Mercury installation for immediate use upon installing the system.
7. Copy the *welcome.py* file in this folder and paste it into a location where you have write permissions on the computer you are using. A good option if you are unsure about this on Windows would be C:\Users\[username]\scripts\, with [username] replaced by whatever your user name is on the machine (note: you will need to create the *scripts* folder). At the same time, also copy the file named *mercury_interface.py* from the Mercury installation directory to your location where you have write permissions. Note that the mercury_interface.py script will not appear in the Mercury menu – this is intentional as this is a helper script that is not meant to be run on its own, so it is automatically hidden.
8. Now we are going to configure a user-generated scripts location in Mercury. To do this, from the top-level menu, choose **CSD Python API**, and then select **Options** from the resulting drop-down menu. Click on the **Add Location** button, browse to the folder where you just saved the copy of the *welcome.py* script and click on **Select Folder**. This will register the folder as an additional source of scripts that Mercury will add to the **CSD Python API** menu.



Location of built-in scripts: C:/Program Files (x86)/CCDC/CSD_2016/mercury 3.8/scripts

Additional Mercury Script Locations

C:/Users/wood/scripts

Settings

Output Directory C:/Users/wood

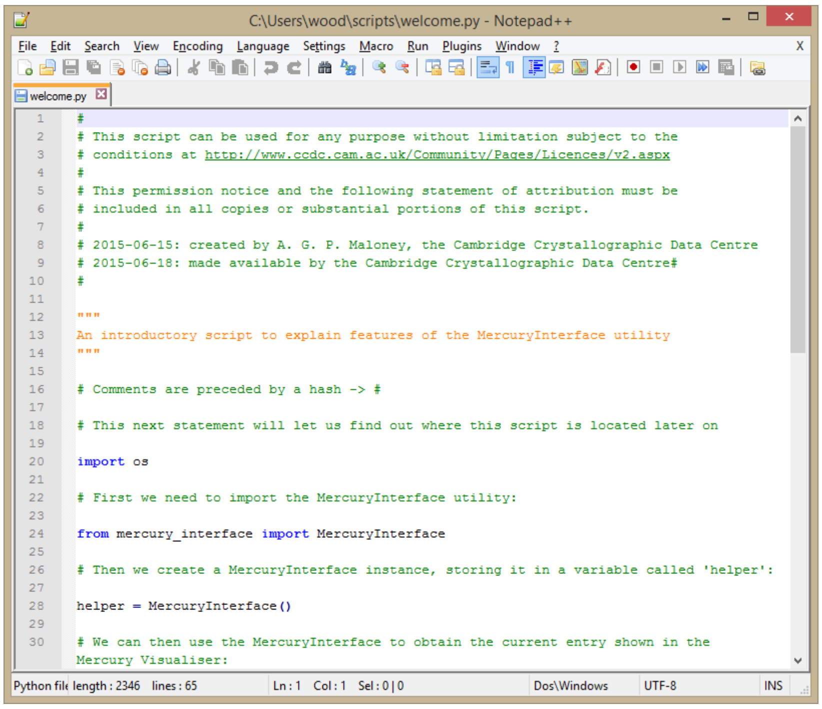
Python Installation

C:/Python27

Python Version: 2.7.10

CSD Python API Version: 1.0.0

9. Now go to the **CSD Python API** top-level menu and you should see that there is a new section in the drop-down menu, listing user-generated scripts, with an item for your copy of the *welcome.py* script. Click on the copy of the *welcome.py* script in your user scripts area of the menu. In the output you will see that the location of the script now matches your user-generated scripts folder location.
10. We are now going to make some edits to the python script to display some additional information about the structure on display. To edit the python script, right-click on the copy of *welcome.py* in your user folder and open it in your text editor.
11. Many of the lines in this script are comments (all those starting with '#') to help explain how the script works and how the interaction between Mercury and the CSD Python API works. You should see a number of references to a helper function called MercuryInterface.


C:\Users\wood\scripts\welcome.py - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
welcome.py

```
#
# 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.
#
# 2015-06-15: created by A. G. P. Maloney, the Cambridge Crystallographic Data Centre
# 2015-06-18: made available by the Cambridge Crystallographic Data Centre#
#

"""
An introductory script to explain features of the MercuryInterface utility
"""

# Comments are preceded by a hash -> #

# This next statement will let us find out where this script is located later on

import os

# First we need to import the MercuryInterface utility:

from mercury_interface import MercuryInterface

# Then we create a MercuryInterface instance, storing it in a variable called 'helper':

helper = MercuryInterface()

# We can then use the MercuryInterface to obtain the current entry shown in the
Mercury Visualiser:
```

```
Python file length : 2346   lines : 65   Ln : 1   Col : 1   Sel : 0 | 0   Dos\Windows   UTF-8   INS
```

12. At the bottom of the script are a series of lines that write the HTML output, each of these lines use the python command to write to a given file: '*f.write()*'. Look for the line including the instruction '*f.write(helper.identifier)*' – this writes to the output file the identifier for the CSD entry, which in this case is '*AABHTZ*'.
13. Copy this line along with the lines both directly before and after it, and then paste the group of three lines again underneath the original group of three in the python file. Edit the text within the brackets as shown below – this will output an additional line of text as well as reporting both the formula relating to the CSD entry and the chemical name.

```
f.write('This is the identifier for the current structure: ')
f.write(entry.identifier)
f.write('<br>')
f.write('These are some extra parameters for the current structure: ')
f.write(entry.formula + ' ' + entry.chemical_name)
f.write('<br>')
```

14. In the welcome.py script, we have already accessed the *entry* object for our structure, in this case the CSD entry AABHTZ. Here we are editing the script to simply read out some further attributes of the entry, namely the chemical formula and the chemical name. If you want to see what other attributes an *entry* object has, take a look at the CSD Python API on-line documentation by choosing **CSD Python API** from the Mercury top-level menu, and then selecting **CSD Python API Documentation** from the resulting drop-down menu.
15. Now re-run the *welcome.py* script from the user-generated scripts section of the **CSD Python API** top-level menu. You will see in the HTML output the additional text and variables relating to the edits that we made to the script.

Conclusions

The initial python script that we ran was copied into a user-generated scripts location and edited to add further functionality to it. Mercury allows multiple user-generated script locations and scripts saved in these areas can be called directly from the menus in the program.

The concept of an *entry* was illustrated here along with some of the attributes that an entry has such as identifier, formula and chemical name. An *entry* also contains a *crystal* attribute, from which further information can be extracted and analyses performed.

You should now know how to run a CSD Python API script from within Mercury as well as how to customise a script and manage user-generated scripts in Mercury.

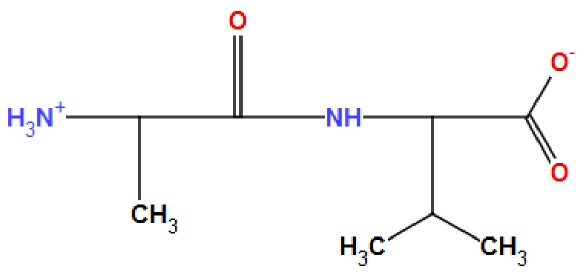
Case Study 2: Generating a simple structural analysis report

Aim

This case study will focus on generating a simple structural analysis report for a known crystal structure. We will introduce the three main types of structural object that the CSD Python API recognises – entries, crystals and molecules – and explain how these objects interact with each other in a hierarchical manner.

Example system

The example system we will be looking at for this case study is L-alanyl-L-valine (shown below) which is a dipeptide compound that has been studied in the crystalline state by Carl Henrik Görbitz. The pure crystal structure of the compound can be found within the Cambridge Structural Database (CSD) under the CSD refcode XUDVOH.



CSD refcode XUDVOH chemical diagram

Analysis

1. Launch Mercury by clicking its icon . From the top-level menu, choose **CSD Python API**, and then select **Options…** from the resulting drop-down menu.
2. Check in this *Mercury Scripting Configuration* dialogue that you have added an additional Mercury script location on your computer for user-generated scripts. If you do not already have a location added here, create a folder for your scripts on your computer (for example in your default user area) and add it to the list in this dialogue by clicking on the **Add Location** button. Click on **Save** to confirm the set of locations and close the dialogue.
3. Now browse to the location you have defined for user-generated Mercury scripts and create a new file in that location named *simple_report.py*. Open up the new python script that you have just created in a text editor such as Notepad, Notepad++ or PyCharm. Ensure that you have also copied the file *mercury_interface.py* from your Mercury installation directory (in the *scripts* folder) to this user-generated scripts location.
4. The python script we have just created is currently blank so we will start with a few simple instructions to allow information to be passed from Mercury to the script and to extract some information from the CSD about the structure we will be analysing.

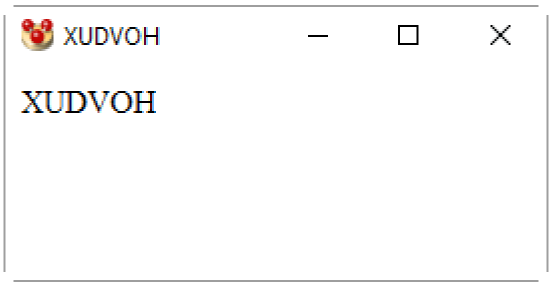
5. Add the following snippet of python to your blank script and save the file:

```
from mercury_interface import MercuryInterface
helper = MercuryInterface()

html_file = helper.output_html_file
entry = helper.current_entry
crystal = entry.crystal
molecule = crystal.molecule

f = open(html_file, "w")
f.write(entry.identifier)
f.close()
```

6. In the first two lines we import the *MercuryInterface* module, from the *mercury_interface.py* script which contains a set of standard functions allowing us to interact with Mercury and then create a *helper* object which enables us to make use of information from Mercury.
7. We next extract the location of our HTML output file from the *helper* object, which will form the basis of the simple structural analysis report, and we then create the structural objects *entry*, *crystal* and *molecule* from the structure currently visualised in Mercury (*current_entry*). Note we haven't defined which structure is being used – whatever structure we have loaded into Mercury will be analysed including CSD entries, in-house database entries and structural files such as MOL2, CIF or RES.

[The objects *entry*, *crystal* and *molecule* are hierarchical in that an *entry* contains a *crystal* and a *crystal* contains one or more *molecules*. An *entry* object contains a variety of details such as experimental conditions like temperature and pressure, bibliographic details and also a *crystal* object. A *crystal* object includes detailed crystal information and functions such as intermolecular interaction details, disorder and void volume. A *molecule* contains further molecule-specific parameters and functions such as number of rings, geometric parameters and the ability to generate SMILES strings. For more details on any of these, see the CSD Python API on-line documentation: http://www.ccdc.cam.ac.uk/docs/csd_python_api/]

8. Finally, we open the output HTML file, write the entry identifier to the file as plain text and close the file.
9. Restart Mercury now to ensure that the user-generated scripts location and the new script will be picked up by Mercury. In the *Structure Navigator* toolbar, type XUDVOH to bring up the structure we are going to analyse.
10. From the top-level menu, choose **CSD Python API**, and then select **simple_report.py** from the resulting drop-down menu. You will see that the HTML file that we've output is automatically read and displayed by Mercury and that the report simply contains the CSD entry identifier.



XUDVOH

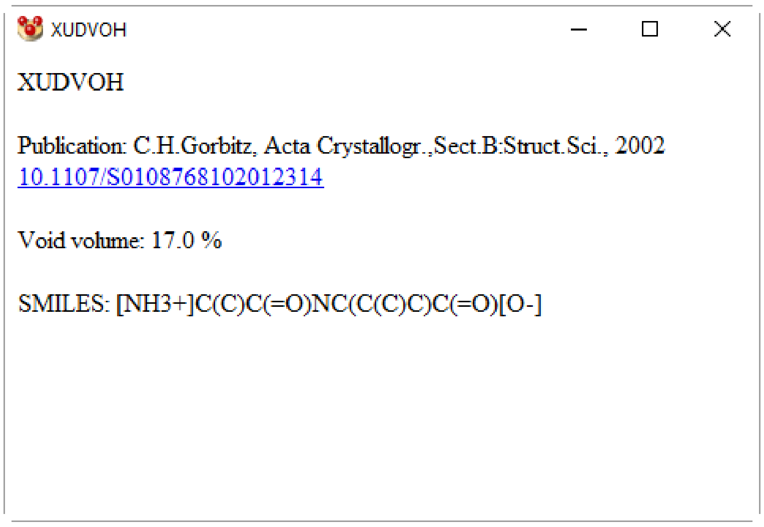
11. Next we will develop the python script to report more detailed information to the user about various aspects of this database entry, crystal and molecule. Begin by adding the following snippet of code just after the definition of the molecule variable and just before we open the HTML file for writing (from line 8).

```
publ = entry.publication
void = crystal.void_volume()
smiles = molecule.smiles
```

12. From the database *entry* object we are extracting information about the article this crystal structure was published in – this will be returned as a **tuple** of (authors, journal, volume, year, first_page, doi). From the *crystal* object we are performing a calculation to determine the void space in the crystal – the void volume as a percentage of the unit cell volume will be returned as a floating point value. Finally, from the *molecule* object we are extracting a SMILES representation – this will be returned as a string. Save the python script again. You can re-run the script from Mercury at this point to check that the script syntax is still okay, but there will be no change to the HTML output yet.
13. Now add the following lines to the script directly above the final line which should read "f.close()". This will output to the HTML file a nicely formatted version of the details that we have extracted from the *entry*, *crystal* and *molecule* objects.

```
f.write('<br><br>')
f.write('Publication: %s, ' % publ.authors)
f.write('%s, %s ' % (publ.journal_name,publ.year))
f.write('<a href="http://dx.doi.org/%s">%s</a>' % (publ.doi,publ.doi))
f.write('<br><br>')
f.write('Void volume: %.1f %%' % void)
f.write('<br><br>')
f.write('SMILES: %s' % smiles)
```

14. Re-run the *simple_report.py* script from the Mercury top-level menu. You will see that the script is returning a formatted report containing useful details about the structure. The output is currently in HTML format so we could further improve the display of the report as much as we liked making use of all the styling options within HTML. There are even modules within python to automatically convert HTML output into Word documents for further ease of use.


XUDVOH

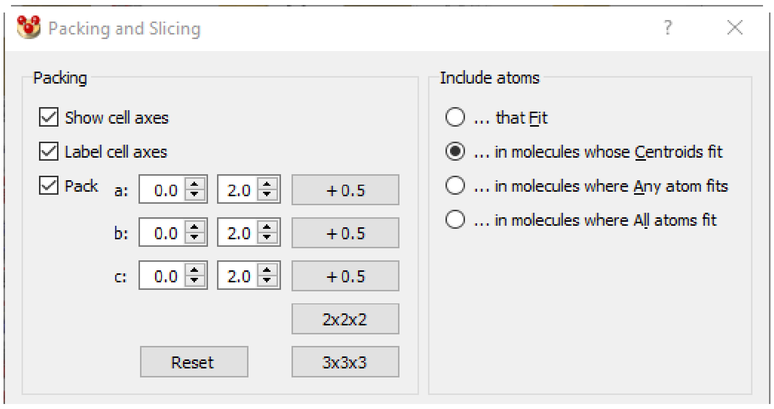
XUDVOH

Publication: C.H.Gorbitz, Acta Crystallogr.,Sect.B:Struct.Sci., 2002
10.1107/S0108768102012314

Void volume: 17.0 %

SMILES: [NH3+]C(C)C(=O)NC(C(C)C)C(=O)[O-]

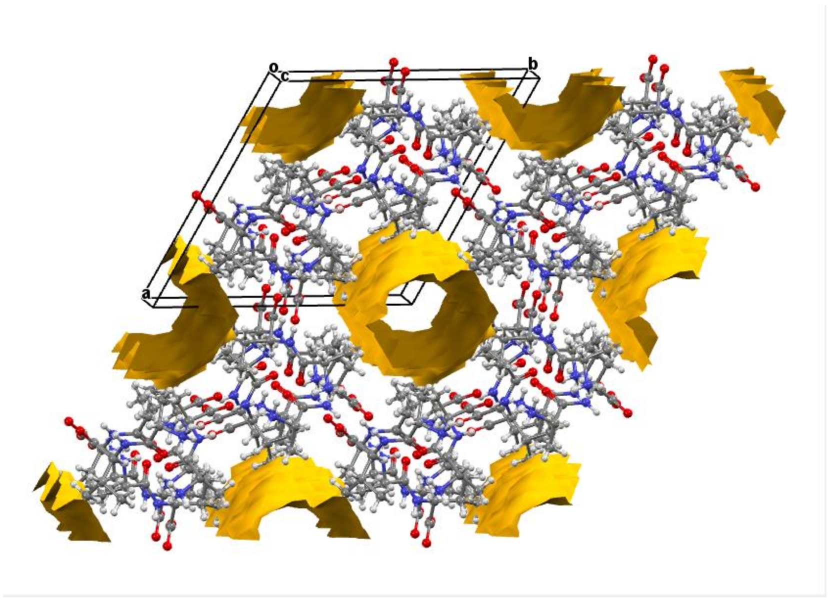
15. The structure that we are studying in this example was published by Carl Henrik Görbitz in 2002 as you can see from the report. It turns out from the void analysis that this structure contains significant void space (17%), which is quite unusual – typical organic structures contain no void space and gaps in the structure of this size are often the result of missed solvent molecules. In this case, however, the structure happens to be an unusually stable nanotube system with flexible pores as you can see from the paper by clicking on the DOI link. The stability of the pores is as a result of very strong, charge-assisted hydrogen-bonding – you can see from the SMILES string that the molecule is zwitterionic with positively and negatively charged groups.
16. To show a slightly large section of the crystal packing, choose **Calculate**, and then select **Packing/Slicing** from the resulting drop-down menu. Ensure that the checkbox labelled **Pack** in this dialogue is toggled on and then click on the button labelled **2x2x2**.


Packing and Slicing

Packing
[✓] Show cell axes
[✓] Label cell axes
[✓] Pack a: 0.0 2.0 +0.5
b: 0.0 2.0 +0.5
c: 0.0 2.0 +0.5

Include atoms
○ ... that Fit
● ... in molecules whose Centroids fit
○ ... in molecules where Any atom fits
○ ... in molecules where All atoms fit

17. To visualise this void space, choose **Display**, and then select **Voids…** from the resulting drop-down menu. Click on the **OK** button to calculate voids and dismiss the dialogue. If you select **Display** from the top-level menu, and then pick **View along** followed by **c**, the channels should be very clear.



Conclusions

Writing this script for use within Mercury has illustrated how simple reports relating to structures can be built up very easily and customised for the information you are interested in. The script we have written highlights the hierarchy and the information contained within the *entry*, *crystal* and *molecule* objects as used by the CSD Python API. Much more elaborate analyses and more finely styled and formatted reports can be produced using this system.

There are a range of other output options that can be harnessed using the mechanism of Mercury's CSD Python API menu. We have seen how HTML files output by scripts run from Mercury will be automatically displayed, but we can also return a new or edited structural file for automatic display in Mercury, a list of results as a GCD file or a set of text/numeric data for display in Mercury's Data Analysis module. For ideas on how to do this, see the other scripts bundled with Mercury or the *mercury_interface.py* helper script.

You should now know how to write CSD Python API scripts from scratch to be run from within Mercury as well as how the *entry*, *crystal* and *molecule* hierarchy works within the CSD Python API.

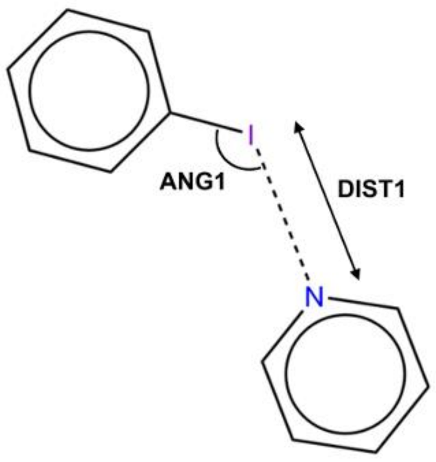
Case Study 3: Searching the CSD for specific interactions

Aim

This case study will focus on using the CSD Python API to carry out a substructure search across the CSD. We will learn how to define substructures, how to apply search settings and constraints, and then how to visualise the data graphically.

Example system

In this example we will investigate the interaction geometry of an aromatic iodine and the nitrogen atom of a pyridine ring. We wish to know if the C-I···N angle tends towards 180° as the I···N distance becomes shorter. Figure 1 illustrates the substructure that we will search the CSD for, with the relevant geometric parameters indicated.



The halogen bonding substructure with defined geometric parameters

Analysis

1. Open your preferred text editor and create a new python file called *interaction_search.py* file that we will run from a command line later on. The following steps show the code that you should write in your Python file, along with explanations of what the code does.

2. We will start by importing the necessary modules for carrying out the substructure search and visualising the data:

```
import ccdc.search
import matplotlib.pyplot as plt
```

   In order to perform a substructure search, we must import the `ccdc.search` module. Additionally, the `matplotlib.pyplot` module will allow us to generate plots to visualise our results. We declare `matplotlib.pyplot as plt` in order to save us a lot of typing later on!

3. There are a number of ways that we can define our substructure, but for this example we will make use of SMARTS strings:

```
ar_I_sub = ccdc.search.SMARTSSubstructure('Ic1ccccc1')
pyridine_sub = ccdc.search.SMARTSSubstructure('n1ccccc1')
```

   Here, `ar_I_sub` specifies the aromatic iodine substructure, and `pyridine_sub` specifies the pyridine substructure, with respective SMARTS strings of **`Ic1ccccc1`** and **`n1ccccc1`**. Note that our atoms of interest, I and N, are both at **index 0** of the SMARTS strings we have defined. If you are unfamiliar with SMARTS strings, you can visualise them and learn more about the format with SMARTSviewer (http://smartsview.zbh.uni-hamburg.de/).

4. We then create our substructure search, which we will call `halogen_bond_search`:

```
halogen_bond_search = ccdc.search.SubstructureSearch()
```

   and add the substructures that we created in the previous step:

```
ar_I_sub_id = halogen_bond_search.add_substructure(ar_I_sub)
pyridine_sub_id = halogen_bond_search.add_substructure(pyridine_sub)
```

   We have added our substructures in this way, giving them identifiers, so we can add our geometric constraints later.

5. We can also specify various criteria for searches by changing the search settings. We can do this in the following way:

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

   This will change certain settings of our `halogen_bond_search`. Here, we have specified that we only wish to search the CSD for organic structures with no disordered atomic positions and a crystallographic *R*-factor of 5.0 or less.

6. We will now apply geometric constraints to our substructure search to limit our search to structures which display characteristic halogen bonding interactions. We will first specify our distance constraint (**DIST1** in Figure 1):

```
halogen_bond_search.add_distance_constraint('DIST1',
                                            ar_I_sub_id, 0,
                                            pyridine_sub_id, 0,
                                            (0.0, 3.4),
                                            'Intermolecular')
```

   Here we have defined an intermolecular distance, DIST1, between the atom at **index 0** of our aromatic iodine substructure (the iodine atom) and the atom at **index 0** of our pyridine substructure (the nitrogen atom). Additionally, we have specified that this distance must be between 0.0 and 3.4 Å.

Similarly, we can specify our angle constraint (**ANG1** in Figure 1):

```
halogen_bond_search.add_angle_constraint('ANG1',
                                         ar_I_sub_id, 1,
                                         ar_I_sub_id, 0,
                                         pyridine_sub_id, 0,
                                         (120.0, 180.0))
```

Here we have defined an intermolecular C-I···N angle and specified that it must lie between 120.0° and 180.0°

7. We are now ready to perform our substructure search. To avoid bias by picking multiple observations from the same structure we will limit the number of hits per structure to 1:

   ```
   halogen_bond_hits = halogen_bond_search.search(max_hits_per_structure=1)
   ```

   This will perform the substructure search, which should take less than a minute. The results from the search will be stored in the **variable** `halogen_bond_hits`.

8. We can now extract our data from halogen_bond_hits using **list comprehension** and Python's built in **zip** functionality:

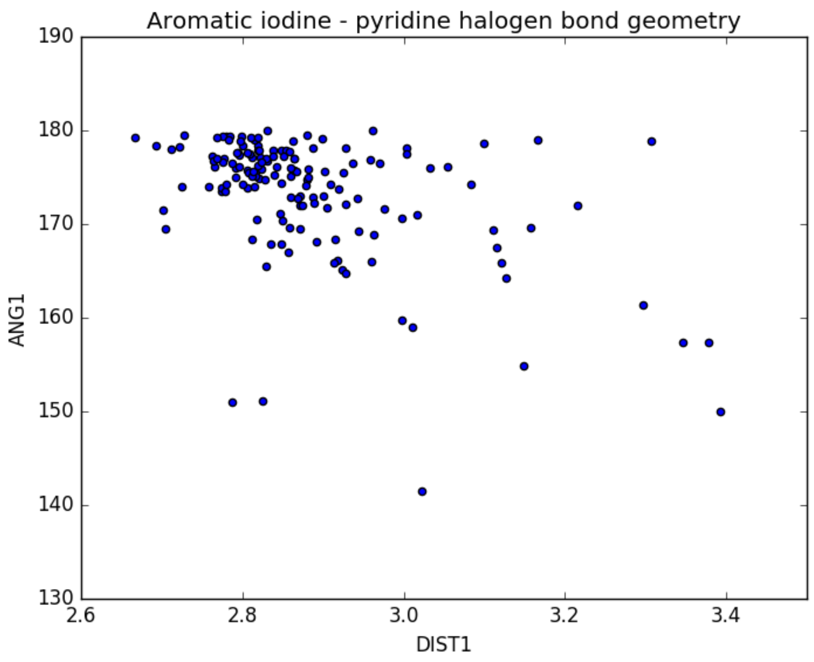
   ```
   dist1_ang1 = [(h.constraints['DIST1'],
                  h.constraints['ANG1']) for h in halogen_bond_hits]

   dist1, ang1 = zip(*dist1_ang1)
   ```

   This will convert our data into a format that will allow us to easily plot DIST1 against ANG1. If you would like more information about Python's built in **`zip`** function, please visit https://docs.python.org/2/library/functions.html#zip.

9. We are now ready to plot our data using the **`scatterplot`** function from **`matplotlib`**:

   ```
   plt.scatter(dist1, ang1)
   plt.title('Aromatic iodine - pyridine halogen bond geometry')
   plt.xlabel('DIST1')
   plt.ylabel('ANG1')
   plt.show()
   ```

   Run the *interaction_search.py* script now from the command line. Here we have plotted DIST1 against ANG1 in a scatterplot, and we have added titles to the plot itself as well as the axes. plt.show() should result in something similar to the following scatterplot being shown:



Conclusions

The substructure search has allowed us to investigate the variation between I···N distance and C-I···N angle in intermolecular halogen bonds between aromatic iodine and pyridine nitrogen. The plot we have generated reveals that there is a weak negative correlation between these parameters – as the contact distance becomes shorter the angle tends towards 180°.

The concept of substructure searching was illustrated here, along with search setting and constraints. Additionally, we have covered how to generate scatterplots as well as some advanced Python functionality.

There are several other ways to perform substructure searches, as well as several different search types, available in the CSD Python API that can be used to answer a number of complex scientific questions.

You should now know how to use the CSD Python API to define a substructure search as well as how to specify additional geometric and search criteria.

Case Study 4: Tackling a scientific challenge using python scripting

Aim

The best way to learn either a scripting language, or a new programmatic interface, is to apply them to a real scientific problem. This case study will aim to test your working knowledge of python scripting and the CSD Python API by setting a real scientific challenge for you to answer. In each case the problem is addressable using only a standard installation of python along with the CSD Python API, but could be tackled in a number of different ways.

Approach

Select one of the scientific challenges laid out below which appeals to you, perhaps something that is aligned with your research area or simply something that interests you as a scientist in general. Making use of the CSD Python API Documentation, along with the hints provided and help from your fellow workshop attendees, write a bespoke python script to address the challenge chosen.

Scientific Challenges

1. Is there a noticeable preference for odd or even numbers of carbons in a molecule? To answer this, plot a graph of the frequency of occurrence of molecules in the CSD as a function of the number of carbons in the molecule. Hint – you may find it easiest to iterate over the whole CSD entry by entry and then iterate over the atoms in a molecule. Note that you probably also want to use only the heaviest molecule per structure.
2. Is there a greater likelihood of significant void space in a crystal structure within some space groups rather than others? To assess this, determine the median void space per structure as a function of the space group number. Hint – void space can only be calculated from the crystal object, using the space group number will help to avoid confusions around space group symbols (for example, $P2_1/c$ is the same as $P2_1/n$, just a different setting).
3. Which first row transition metals display the highest prevalence for Jahn-Teller distortion? To answer this, plot histograms of the metal-ligand bond lengths for each first row transition metal and compare the distributions. Hint – `len(atom.neighbours)` can be used to find out the coordination number of the metal atoms. Focus only on octahedral complexes for this example.
4. ROY (CSD Refcode QAXMEH) is a notoriously polymorphic compound. Using the Conformer Generator, explore the conformational space of this compound by generating a number of conformers and comparing them with a Packing Similarity calculation. Is this compound likely to exhibit conformational polymorphism? Hint – you may want to take a look at the overlay function within ccdc.descriptors.MolecularDescriptors and utilise RMSD. A high RMSD value for the similarity is an indicator of conformational polymorphism.

Conclusions

You should now be confident in writing scripts based around the CSD Python API to tackle genuine scientific problems, as well as have a good working knowledge of the extent of the CSD Python API.