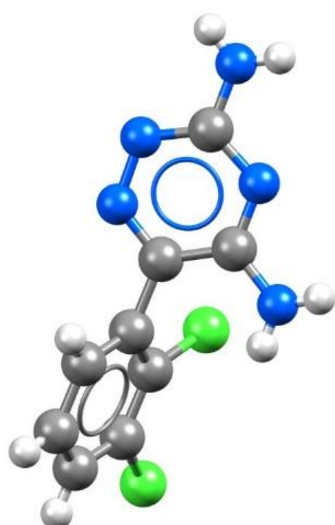
The CSD Python API: Advanced Research Applications

2020.0 CSD Release
CSD Python API version 3.3.0



CCDC
advancing structural science

Introduction

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 CSD Python API, building from an initial introduction to the basic mechanics of input and output through a Python console, to the CSD Python API menu in Mercury, and finally 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, as well as a very basic understanding of Python. If you require any further assistance with any aspect of what is contained within this workshop material, please do not hesitate to ask your workshop instructor, or consult the online Python documentation at https://docs.python.org/3.7/ or the help guides included within Mercury.

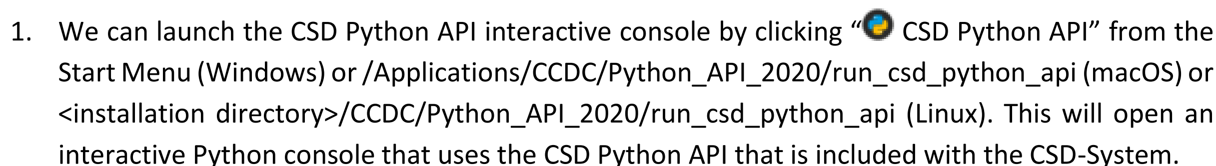
CSD Python API scripts can be run from the command-line or from within Mercury to achieve a wide range of analyses, research applications and generation of automated reports

Case Study 1: Demonstrating Input and Output

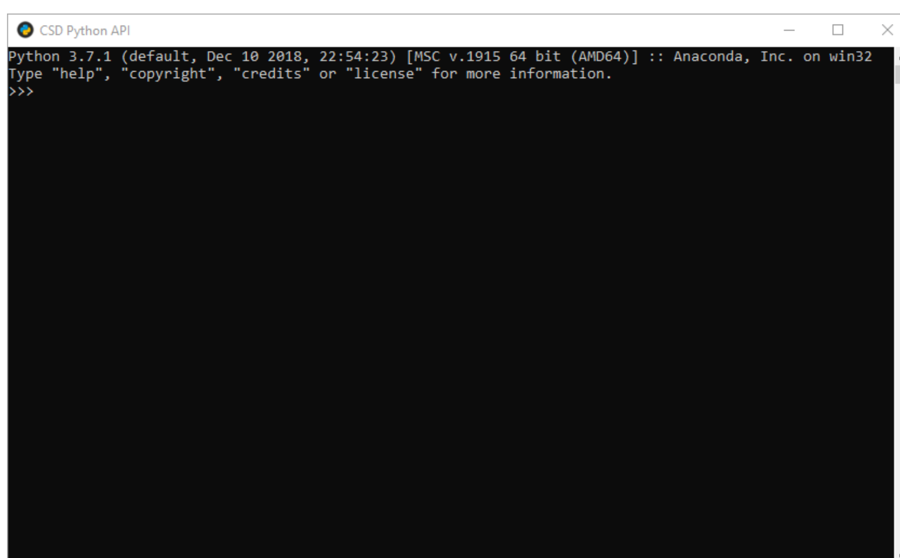
Aim

This case study will focus on understanding the basic principles of using the CSD Python API. We will use the CSD Python API interactive console to learn about input and output, and we will cover the concepts of Entries, Molecules and Crystals.

Instructions



1. We can launch the CSD Python API interactive console by clicking " CSD Python API" from the Start Menu (Windows) or /Applications/CCDC/Python_API_2020/run_csd_python_api (macOS) or <installation directory>/CCDC/Python_API_2020/run_csd_python_api (Linux). This will open an interactive Python console that uses the CSD Python API that is included with the CSD-System.



2. When we see ">>>" in the console window, it means that Python is ready for a command. We will type all of our code after these ">>>" and hit "Enter" to send these instructions to Python.

3. The CSD Python API makes use of different modules to do different things. The ccdc.io module is used to read and write entries, molecules and crystals. To make use of modules, we first need to import them. Type the following into the Python console then hit "Enter":

   ```
   from ccdc import io
   ```

   After a few seconds, the ">>>" will appear again, indicating that we have successfully imported the ccdc.io module. We will now explore some of what this module allows us to do.

4. Entries, molecules and crystals are different types of Python objects, and have different characteristics, although they do have a number of things in common. They each have readers and writers that allow for input and output respectively. We will start by setting up an entry reader and using it to access the CSD. Type the following into the Python console then hit "Enter" after each line:

```
entry_reader = io.EntryReader('CSD')
first_entry = entry_reader[0]
print(first_entry.identifier)
```

You should see that Python has returned “AABHTZ”, which is the identifier of the first entry in the CSD. Giving the 'CSD' argument to the EntryReader will open the installed CSD database, and it is possible to open alternative or multiple databases in this way. Similar methods can be used to read molecules or crystals with a MoleculeReader or CrystalReader instance.

5. From an entry object, it is also possible to access the underlying molecule or crystal for that CSD entry. We will explore this using paracetamol (CSD Refcode HXACAN). Type the following into the Python console then hit “Enter” after each line (Note – the four spaces before print are very important!):

```
hxacan_entry = entry_reader.entry('HXACAN')
hxacan_molecule = hxacan_entry.molecule
for atom in hxacan_molecule.atoms:
    print(atom.label)
```

You should see that Python has returned the label for each atom in the paracetamol molecule. What we are doing here is accessing the entry HXACAN directly from our EntryReader, then accessing the underlying molecule from this entry. We then use a for loop to iterate through each atom in the molecule and print out its atom label. for loops are used to iterate through each item in a list of items – the atoms in the molecule in this case. for loops are really useful, and allow us to iterate through everything from the atoms in a molecule to entries in the CSD.

6. We can also read entries, molecules and crystals from a number of supported file types. We are going to use an example *.cif* file to illustrate this. For this demonstration, we will use the provided *example.cif* (which you can access here) and place it into a folder that we have read and write access to, for example C:\training\ for Windows, or something equivalent on macOS or Linux.

   We need to tell Python where to find this file, so type the following into the Python console then hit “Enter”, making sure that the filepath is that which you have just used:

```
filepath = r"C:\training\example.cif"
```

   Python doesn’t like spaces or backslashes in file paths! The r and double quotes (" ") help us to get around this.

7. Now that Python knows where our *.cif* file is located we can access the crystal using a CrystalReader, so type the following into the Python console then hit “Enter” after each line:

```
crystal_reader = io.CrystalReader(filepath)
tutorial_crystal = crystal_reader[0]
print(tutorial_crystal.spacegroup_symbol)
```

   The Python console should display the spacegroup of our example crystal, $P2_1/n$. The 0 means that we want to access the first crystal in the .cif (when we have multiple items in a list or a file, Python starts numbering them from zero).

8. It is good practice to close files when we are finished with them, but before we do that, we are going to take the underlying molecule from our tutorial crystal for use later. Type the following into the Python console then hit "Enter" after each line:

```
tutorial_molecule = tutorial_crystal.molecule
crystal_reader.close()
```

9. The CSD Python API can also write entries, molecule and crystals to a number of supported file types. To do this, we need to tell Python where we want the file to be written. We will continue to use our C:\training\ folder (or equivalent), and we will use this to set up our new file as a variable. To do this, type the following into the Python console hitting "Enter":

```
f = r"C:\training\mymol.mol2"
```

10. With this new variable we can use the CSD Python API to create a *.mol2* file that contains the molecule from the example *.cif* file that we kept from earlier. To do this, type the following into the Python console then hit "Enter" after each line:

```
with io.MoleculeWriter(f) as mol_writer:
    mol_writer.write(tutorial_molecule)
```

    Here, the **with** statement ensures that we automatically close the mol_writer and the file when we have written our molecule.

    What we have done in this last step is to create a file *mymol.mol2* in our folder, then write the molecule we kept from earlier into it. In this way, we can write out molecules, crystals and entries that we have obtained or modified and use them for other tasks and with other programs.

Conclusions

The CSD Python API was used with the built-in Python console to explore input and output of various objects and file types using the ccdc.io module.

The concepts of an *entries, molecules* and *crystals* were illustrated here along with some of the ways in which these are related.

You should now know how to run the CSD Python API from a Python console, and have an appreciation of how objects and files are read into and written out of the CSD Python API.

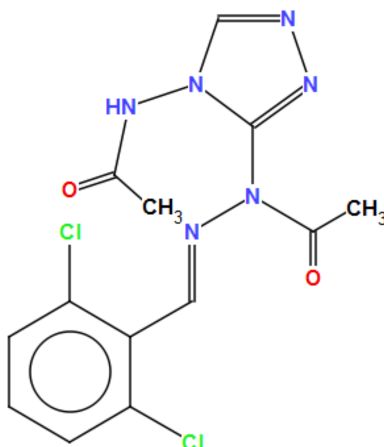
Case Study 2: Customising a simple script for use in Mercury

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.

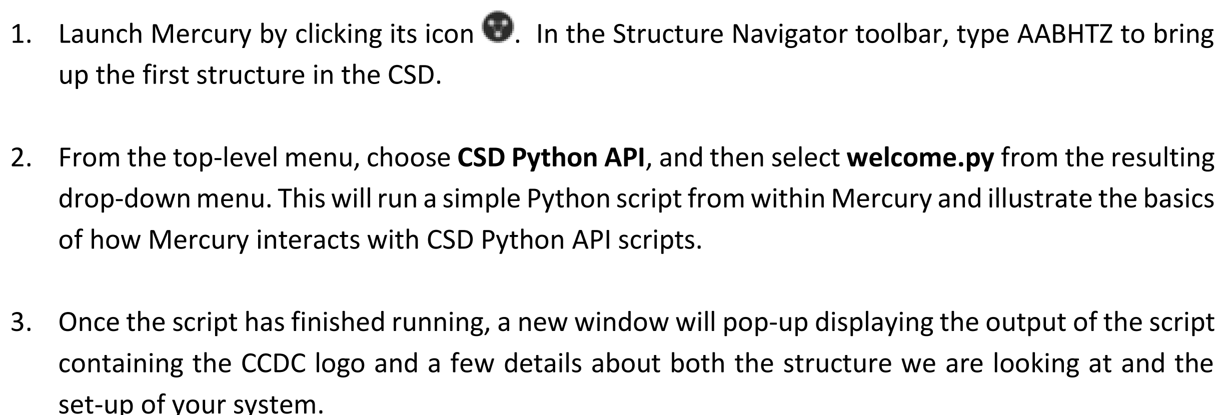
Example system

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 with the CSD refcode AABHTZ.



CSD refcode AABHTZ chemical diagram

Instructions



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.

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.

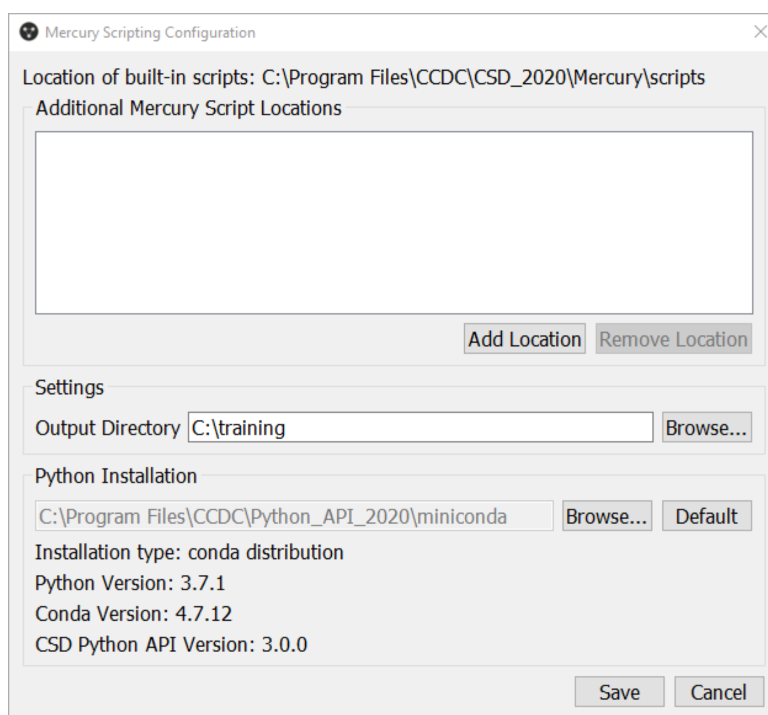
[For small scale editing of text files and Python scripts, such as the edits made in this case study, we would recommend Notepad++ for Windows (https://notepad-plus-plus.org/) and TextWrangler for macOS (available in the App Store). For more in-depth Python editing or for interactive work, try looking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



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 # or surrounded by triple quote marks ”””) 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\pradon\scripts\welcome.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Window ?
welcome.py
1  #
2  # This script can be used for any purpose without limitation subject to the
3  # conditions at http://www.ccdc.cam.ac.uk/Community/Pages/Licences/v2.aspx
4  #
5  # This permission notice and the following statement of attribution must be
6  # included in all copies or substantial portions of this script.
7  #
8  # 2015-06-15: created by A. G. P. Maloney, the Cambridge Crystallographic Data Centre
9  # 2015-06-18: made available by the Cambridge Crystallographic Data Centre#
10 #
11
12 """
13 An introductory script to explain features of the MercuryInterface utility
14 """
15
16 # Comments are preceded by a hash -> #
17
18 # This next statement will let us find out where this script is located later on
19
20 import os
21
22 # First we need to import the MercuryInterface utility:
23
24 from mercury_interface import MercuryInterface
25
26 # Then we create a MercuryInterface instance, storing it in a variable called 'helper':
27
28 helper = MercuryInterface()
29
30 # We can then use the MercuryInterface to obtain the current entry shown in the Mercury Visualiser:
31
32 entry = helper.current_entry
Python file    length : 2,610   lines : 70    Ln : 1   Col : 1   Sel : 0 | 0    Windows (CR LF)   UTF-8   INS
```

12. Starting on Line 41 of the script are a series of lines that provide the *content* to write to the HTML output. Each of these lines uses a mixture of HTML and Python commands to write formatted text to a given file'. Look for the line including the words *helper.identifier* – this writes to the output file the identifier for the CSD entry, which in this case is '*AABHTZ*'.

13. Below this line we will add some more information to the *content* to be displayed when we run the script. Edit the text as shown below – this will output some additional lines of text as well reporting both the formula relating to the CSD entry and the chemical name.

```
'This is the identifier for the current structure: <b>%s</b>' % helper.identifier,
# Add the additional information in here
'This is the chemical formula of the current structure: <b>%s</b>' % entry.formula,
'And the chemical name of the current structure: <b>%s</b>' % entry.chemical_name,
```

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, 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 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.

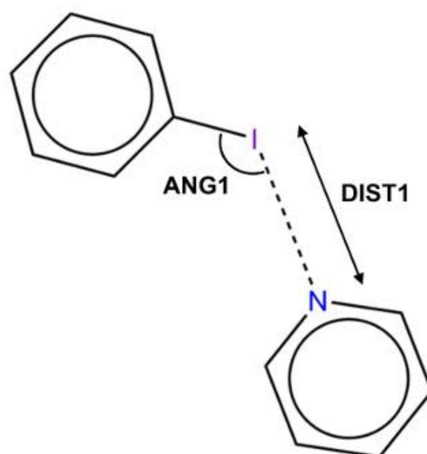


Figure 1: The halogen bonding substructure with defined geometric parameters

Instructions

1. Open your preferred text editor and create a new Python file called *interaction_search.py* that we will run from a command line later. 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 matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import ccdc.search
```

   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` into two separate lists, one for distances and one for angles:

```
dist1 = []
ang1 = []
for h in halogen_bond_hits:
    dist1.append(h.constraints['DIST1'])
    ang1.append(h.constraints['ANG1'])
```

This will convert our data into a format that will allow us to easily plot DIST1 against ANG1.

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()
```

10. To run the *interaction_search.py* script from the command line, you will first need to activate your environment. The activation method will vary depending on the platform:

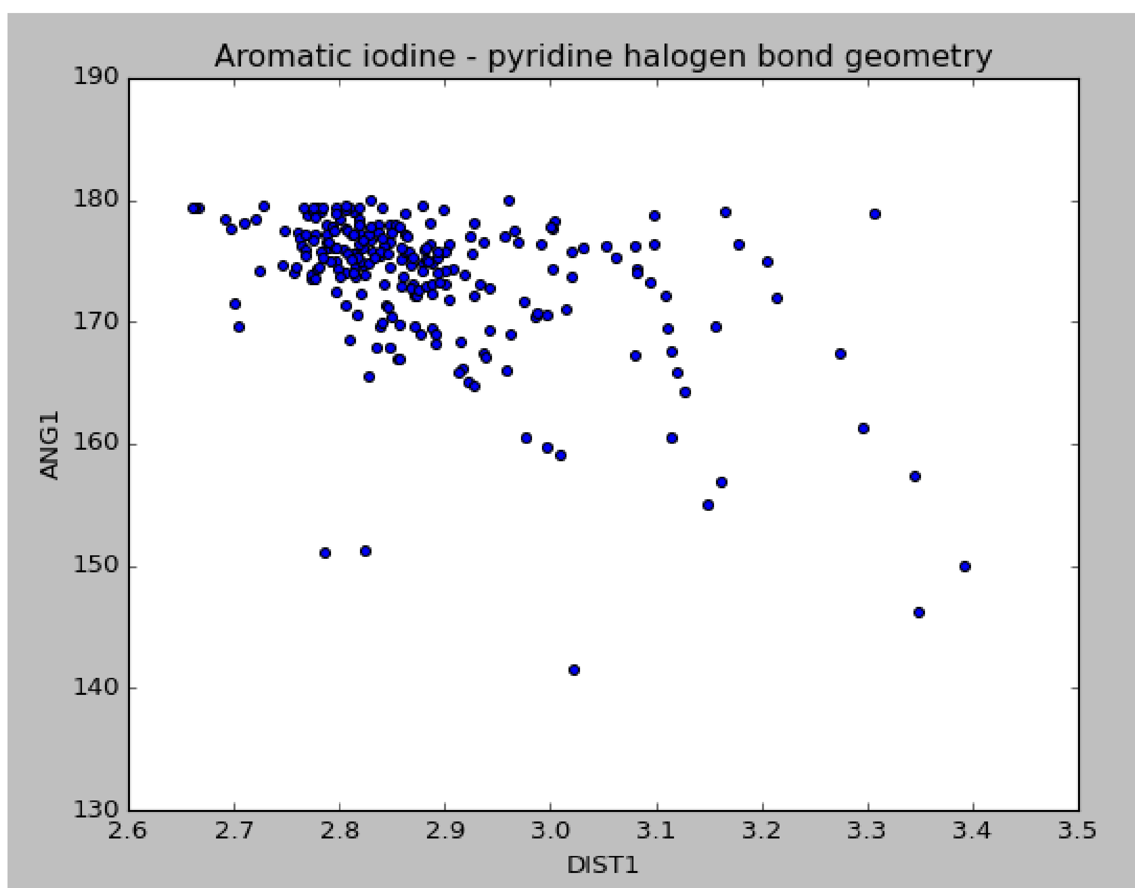
- Windows: Open a Command Prompt window and type:
  "C:\Program Files\CCDC\Python_API_2020\miniconda\Scripts\activate"
- MacOS/Linux: Open a terminal window and change directory to the CSD Python API bin folder:
  cd /Applications/CCDC/Python_API_2020/miniconda/bin

  Then activate the environment with:
  source activate

Once you have activated your environment, change directory to where you saved your script, and run it by typing:

```
python interaction_search.py
```

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 settings 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 many 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: Filtering the CSD to find organic hydrates

Aim

For many purposes, it is often useful to generate subsets of the CSD. This case study shows how to systematically search through the CSD to find a specific class of structure. It also shows how to apply search filters outside of a conventional search operation.

Instructions

1. Open your preferred text editor and create a new Python file called *hydrates_filter.py* that we will run from a command line later. 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 interrogating the CSD and for applying filters:

```
from ccdc import io, search
```

   In order to read the CSD, we must make use of the `ccdc.io` module. To apply filters, we must import the `ccdc.search` module. This is the syntax used if you need to import multiple modules from the same library.

3. We want to specify some criteria that we can use to filter the CSD as we search through it. To do this, we can make use of a `ccdc.search.Search.Settings` instance similar to the one that we used in Case Study 3 above:

```
settings = search.Search.Settings()
settings.only_organic = True
settings.not_polymeric = True
settings.has_3d_coordinates = True
settings.no_disorder = True
settings.no_errors = True
settings.max_r_factor = 7.5
```

   Here, our filter indicates that we only want to return organic, non-polymeric structures that are of high quality (with a crystallographic *R*-factor of 7.5 or less), without disorder and errors.

4. Next, we want to set up a counter to track our search and set up an EntryReader to search the CSD:

```
count = 0
csd = io.EntryReader('CSD')
```

5. We will now set up our output file with an EntryWriter instance and begin iterating through the CSD. It is often useful at this stage of a script to provide some feedback as to how it is progressing:

```
with io.EntryWriter('hydrates.gcd') as writer:
    for i, entry in enumerate(csd):
        if i % 10000 == 0:
            print('Found {} hydrates from {} entries...'.format(count, i))
```

   Remember that the indentations here are important! Using the **with** syntax here means that our *.gcd* file (refcode list) will automatically close when the script is finished. The `%` is the *modulo*

operator, which returns the remainder of dividing one number by another. This block of code will work through each entry in the CSD in turn, and provide feedback for every 10,000 entries it has assessed.

6. The final script will take anywhere from 1 to 2 hours to run depending on the speed of your machine. To reduce the time to a few minutes for this workshop, we'll add the following:

```
# This block is only to save time. The entire check will take 1-2 hours.
if count > 200:
    break
# end block
```

7. Next, we want check that the current CSD entry passes the criteria that we outlined above. Make sure that this next piece of code is lined up under the last **if** statement so that it is part of the correct block:

```
if settings.test(entry):
    molecule = entry.molecule
    hydrate = False
```

   Here we are testing the current entry against the criteria that we specified previously – if an entry passes the test it will return **True** otherwise it will return **False**, and we can use this to decide whether we want to carry out the next instructions or not. If the entry passes our test, we want to start analysing the molecule object. We're also setting a flag at this stage that we'll use later when we're deciding if we have a hydrate or not.

8. We now want to check each component of the molecule object that we're investigating, and check if we have any water molecules present in our structure. Again, make sure to check the indentation of your code – these next lines should line up with the last lines we've written:

```
    for component in molecule.components:
        if component.smiles == 'O' and component.all_atoms_have_sites:
            hydrate = True
```

   Here we are iterating through each component in the current structure and checking its corresponding SMILES string to identify any water molecules. We are also being particularly stringent to make sure that the water molecules that we find have explicit hydrogen atom positions – we want a high-quality data set! If the current entry contains a water molecule that passes our test, we set our hydrate flag to **True** for the next step.

9. We want to write out the refcodes of any hydrated crystal structures that pass our test to our refcode list. The indentation here should line up under the last **for** statement:

```
    if hydrate:
        writer.write(entry)
        count += 1
```

   We now make use of our `hydrate` flag so that we can control which entries are written to the output file. We also add one to our count for each entry that we write so that we can use this count in our feedback loop to keep track of our progress.

10. Finally, at the end, we will print the number of hydrates we have found

```
print('Finished: Found {} hydrates from {} entries.'.format(count, i))
```

Your final script should look like this:

```
from ccdc import io, search

settings = search.Search.Settings()
settings.only_organic = True
settings.not_polymeric = True
settings.has_3d_coordinates = True
settings.no_disorder = True
settings.no_errors = True
settings.max_r_factor = 7.5

count = 0
csd = io.EntryReader('CSD')

with io.EntryWriter('hydrates.gcd') as writer:
    for i, entry in enumerate(csd):
        if i % 10000 == 0:
            print('Found {} hydrates from {} entries...'.format(count, i))

        # This block is only to save time. The entire check will take 1-2 hours.
        if count > 200:
            break
        # end block

        if settings.test(entry):
            molecule = entry.molecule
            hydrate = False

            for component in molecule.components:
                if component.smiles == 'O' and component.all_atoms_have_sites:
                    hydrate = True

            if hydrate:
                writer.write(entry)
                count += 1
    print('Finished: Found {} hydrates from {} entries.'.format(count, i))
```

11. Now run the *hydrates_filter.py* script from the command line. (For information on how to do so, please see Case Study 3, step 10.) It should create a file, *hydrates.gcd*, in the directory you are working in where all the results will be captured. As the script progresses, you should see a feedback message displayed every 10,000 structures indicating how many entries that meet our criteria have been found so far. If you run the complete script by removing the time saving block, you should end up with around 13,000 organic hydrates in your refcode list that you can use for further analysis later.

Conclusions

Setting up filters has allowed us to search the CSD for organic hydrates, which we have captured in a refcode list, or *.gcd* file.

The concept of iterating through entries in a database was introduced here, as well as using search criteria to act as filters. Additionally, we have recapped how to produce output files.

While there are several standard filters that can be applied to searches, to answer more challenging scientific questions using the CSD Python API it is possible to construct bespoke filters that are tailored specifically to your needs.

You should now know how to use the CSD Python API to define criteria for search filters as well as how to iterate through a structural database.

Case Study 5: 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 several 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. When analysing crystal structures that appear to contain small voids, it is often useful to know for reference the volume of space commonly occupied by a water molecule. Remove the block of code starting with "# This block is only to save time." (through the line with "# end block"). Run this script again to generate the full list of hydrates. Using this list, calculate the average volume occupied by a water molecule in the CSD.
   Hint – The following snippet of code shows how to manipulate the underlying molecule of a crystal. Think of how you would identify the water atoms in each hydrated structure.

   ```
   molecule = crystal.molecule
   molecule.remove_atoms(molecule.atom(w.label) for w in water_atoms)
   crystal.molecule = molecule
   ```

2. To create useful subsets of the CSD, you may wish to perform searches based on general descriptions of molecules. Construct a subset of the CSD containing only molecules that have *X* donors and *Y* acceptors, where *X* and *Y* are numbers of your choosing.
   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.
3. 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).

Conclusions

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