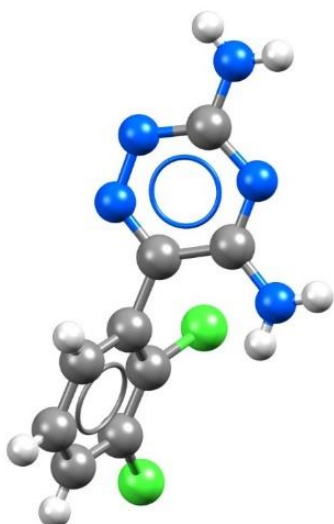


Advanced search and analysis using the CSD Python API (PYAPI-002)

2023.1 CSD Release
CSD Python API version 3.0.15



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.



CCDC
advancing structural science

Table of Contents

Introduction	3
Learning Outcomes	3
Pre-required Skills	3
Materials	3
Activating the command line	4
Example 1: Searching the CSD for specific interactions.....	5
Aim	5
Instructions	5
Conclusions	8
Example 2: Filtering the CSD to find organic hydrates.	10
Aim	10
Instructions	10
Conclusions	12
Example 3: Tackling a scientific challenge using Python scripting.....	14
Aim	14
Approach.....	14
Scientific Challenges.....	14
Conclusions	14
Summary	15
Next Steps	15
Feedback	15
Glossary.....	15

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 workshop will cover advanced searching of specific types of interactions in the CSD. 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).

Before beginning this workshop, ensure that you have a registered copy of CSD-Core or above installed on your computer. Please contact your site administrator or workshop host for further information.

Learning Outcomes

In this workshop, you will learn how to:

- Search for specific interactions in the CSD.
- Save your search results.
- Visualize outputs in a plot.
- Create search filters.

This workshop will take approximately **25** minutes to complete guided Examples 1 and 2. Example 3 is an open challenge and can be completed at your own pace. The words in *Blue Italics* in the text are reported in the [Glossary](#) at the end of this handout.

Pre-required Skills

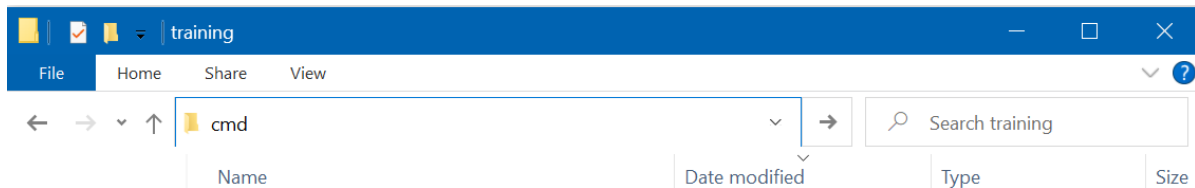
The following exercises assume that you have a working knowledge of Python. We recommend working through the Introduction to the CSD Python API workshop (PYAPI-001) that can be found [here](#) before starting this workshop.

Materials

A text editor is required for scripting during this workshop. If you have a preferred text editor, we recommend sticking with that. If you do not have a preferred editor, we would recommend Notepad++ for Windows (<https://notepad-plus-plus.org/>) and BBEdit for macOS (available in the App Store). The basic Notepad functionalities in Windows would also be enough. For more in-depth Python editing or for interactive work, try looking at PyCharm (<https://www.jetbrains.com/pycharm/>) or Jupyter (<https://jupyter.org/>). Visual Studio is available for all platforms and would be a suitable editor (<https://visualstudio.microsoft.com/downloads/>).

Activating the command line

1. For this exercise we will be writing the script in a Python file that we can then run from a command prompt later. Start by creating a folder where you will save your Python files in a place where you have read and write access, for example C:\training\ for Windows, or something equivalent on macOS or Linux. We will continue to use our C:\training\ folder (or equivalent), through the tutorial.
2. Open the command prompt from this folder. In Windows you can type 'cmd' in the File Explorer tab and press 'Enter'. In Linux you can right click on the folder and select Open in Terminal. In macOS, right click on the folder, select Services then click New Terminal at Folder.

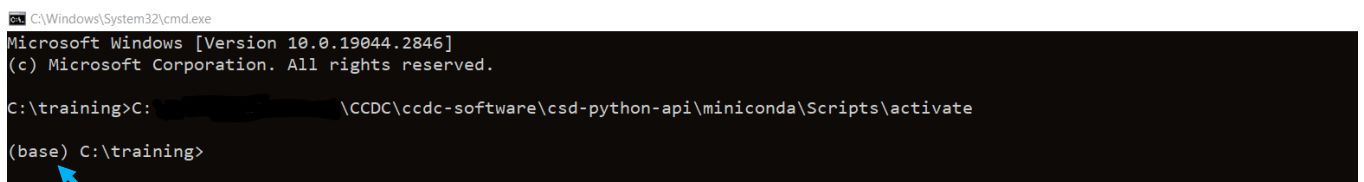


The command prompt window should now appear.



3. To run your Python scripts from the command prompt, 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 (including the " marks):
`"C:\path\to\CCDC\ccdc-software\csd-python-api\miniconda\Scripts\activate"`
 - **MacOS/Linux:** Open a terminal window and change directory to the CSD Python API bin folder:
`cd /Applications/CCDC/ccdc-software/csd-python-api/miniconda/bin`
 Then activate the environment with:
`source activate`

Replace "path\to" with the exact path to your ccdc-software install. If the activation is successful, (base) will appear at the beginning of your command prompt:



Example 1: Searching the CSD for specific interactions.

Aim

This example 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.

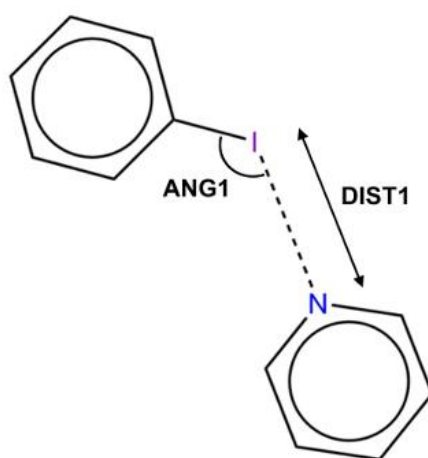


Figure 1. The halogen bonding substructure with defined geometric parameters

Instructions

1. Open your preferred text editor and create a new Python file called *interaction_search.py* that we will run from a command prompt 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` to save us a lot of typing later on!

- There are several ways that we can define our substructure, but for this example we will make use of [SMARTS](#) strings – a way of describing chemical structure using letters, numbers and symbols:

```
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/>).

- 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.

- 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.

- 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 prompt, you will first need to activate your environment. Activation steps are covered [above](#).

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 a scatterplot similar to Figure 2 on the following page.

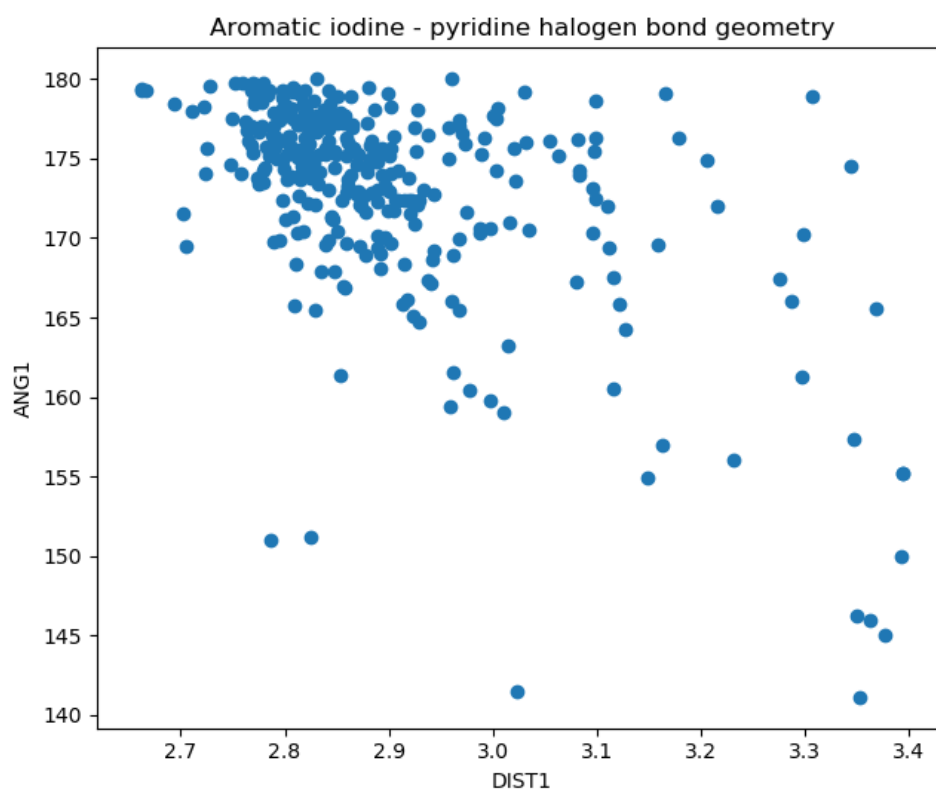


Figure 2. A scatterplot of ANG1 against DIST 1.

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.

Full script

Note: if you copy and paste the script below, double check that the spacing is correct (see image below for indentation).

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import ccdc.search
ar_I_sub = ccdc.search.SMARTSSubstructure('Ic1cccc1')
pyridine_sub = ccdc.search.SMARTSSubstructure('n1cccc1')
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)
halogen_bond_search.settings.only_organic = True
halogen_bond_search.settings.no_disorder = 'all'
halogen_bond_search.settings.max_r_factor = 5.0
halogen_bond_search.add_distance_constraint('DIST1',
                                           ar_I_sub_id, 0,
                                           pyridine_sub_id, 0,
                                           (0.0, 3.4),
                                           'Intermolecular')
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))
halogen_bond_hits = halogen_bond_search.search(max_hits_per_structure=1)
dist1 = []
ang1 = []
for h in halogen_bond_hits:
    dist1.append(h.constraints['DIST1'])
    ang1.append(h.constraints['ANG1'])
plt.scatter(dist1, ang1)
plt.title('Aromatic iodine - pyridine halogen bond geometry')
plt.xlabel('DIST1')
plt.ylabel('ANG1')
plt.show()
```

```
import ccdc.search
ar_I_sub = ccdc.search.SMARTSSubstructure('Ic1cccc1')
pyridine_sub = ccdc.search.SMARTSSubstructure('n1cccc1')
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)
halogen_bond_search.settings.only_organic = True
halogen_bond_search.settings.no_disorder = 'all'
halogen_bond_search.settings.max_r_factor = 5.0
halogen_bond_search.add_distance_constraint('DIST1',
                                           ar_I_sub_id, 0,
                                           pyridine_sub_id, 0,
                                           (0.0, 3.4),
                                           'Intermolecular')
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))
halogen_bond_hits = halogen_bond_search.search(max_hits_per_structure=1)
dist1 = []
ang1 = []
for h in halogen_bond_hits:
    dist1.append(h.constraints['DIST1'])
    ang1.append(h.constraints['ANG1'])
plt.scatter(dist1, ang1)
plt.title('Aromatic iodine - pyridine halogen bond geometry')
```

Example 2: 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 example 1 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.

- 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 will add the following:

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

- 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 are also setting a flag at this stage that we will use later when we are deciding if we have a hydrate or not.

- We now want to check each component of the molecule object that we are 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 have 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.

- 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))
```

11. Now run the `hydrates_filter.py` script from the command line. (For information on how to do so, please see Example 1, 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 at least 17,300 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.

[Full script](#)

Note: if you copy and paste the script below, double check that the spacing is correct (see image below for indentation).

```

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))

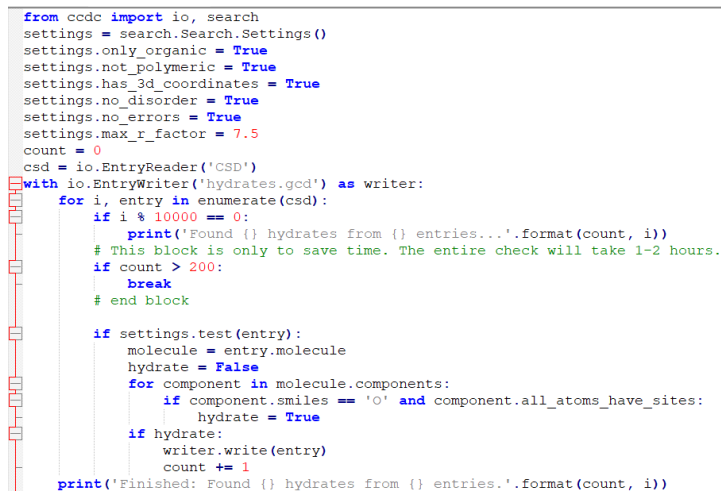
            # This block is only to save time. The entire check will take 1-2
            # hours.
            if count > 200:
                break
            # end block

            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))

```



```

1 from ccdc import io, search
2 settings = search.Search.Settings()
3 settings.only_organic = True
4 settings.not_polymeric = True
5 settings.has_3d_coordinates = True
6 settings.no_disorder = True
7 settings.no_errors = True
8 settings.max_r_factor = 7.5
9 count = 0
10 csd = io.EntryReader('CSD')
11 with io.EntryWriter('hydrates.gcd') as writer:
12     for i, entry in enumerate(csd):
13         if i % 10000 == 0:
14             print('Found {} hydrates from {} entries...'.format(count, i))
15             # This block is only to save time. The entire check will take 1-2 hours.
16             if count > 200:
17                 break
18             # end block
19
20             if settings.test(entry):
21                 molecule = entry.molecule
22                 hydrate = False
23                 for component in molecule.components:
24                     if component.smiles == 'O' and component.all_atoms_have_sites:
25                         hydrate = True
26
27                 if hydrate:
28                     writer.write(entry)
29                     count += 1
30 print('Finished: Found {} hydrates from {} entries.'.format(count, i))

```

Example 3: 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](#), 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 easier 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 have now gained experience 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.

Summary

This workshop introduced the CSD Python API. You should now be familiar with:

- Accessing CSD entries through the CSD Python API.
- Conducting search for specific interactions in the CSD.
- Visualising search outputs in a plot.
- Saving search results in a recode list (*.gcd* file).
- Creating search filters.

Next Steps

You can explore other workshop materials from the CSD-Core workshops section (<https://www.ccdc.cam.ac.uk/community/training-and-learning/workshop-materials/csd-core-workshops/>). If you are interested in continuing learning about the CSD Python API, a ligand-based virtual screening workshop using the CSD Python API can be found in the CSD-Discovery workshops section (<https://www.ccdc.cam.ac.uk/community/training-and-learning/workshop-materials/csd-discovery-workshops/>).

Feedback

We hope this workshop improved your understanding of the CSD Python API and you found it useful for your work. As we aim to continuously improve our training materials, we would love to get your feedback. Click on [this link](#) to a survey, it will take less than 5 minutes to complete. The feedback is anonymous. You will be asked to insert the workshop code, which for this self-guided workshop is PYAPI-002. Thank you!

Glossary

Recode list or ***.gcd* file** – a file containing a list of CSD Recodes. This file can be opened in various CCDC applications.

SMARTS string - a way of describing a chemical substructure using letters, numbers and symbols. 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/>).

SMILES – Simplified Molecular Input Line Entry System; a chemical notation for describing the structure of chemical species using short strings.

Substructure – A substructure is a part or section of a whole molecule. When used in a search using the CSD Python API, the substructure can be identified within other molecules.