rcsbsearchapi

Release 1.5.1

Dennis Piehl

Feb 19, 2024

CONTENTS

The `rcsbsearchapi` package provides a Python interface to the RCSB PDB Search API. Use it to fetch lists of PDB IDs corresponding to advanced query searches.

CHAPTER

ONE

QUICKSTART

1.1 Installation

Get it from PyPI:

```
pip install rcsbsearchapi
```

Or, download from GitHub

1.2 Syntax

Here is a quick example of how the package is used. Two syntaxes are available for constructing queries: an "operator" API using python's comparators, and a "fluent" syntax where terms are chained together. Which to use is a matter of preference.

A runnable jupyter notebook with this example is available in notebooks/quickstart.ipynb, or can be run online using binder:

An additional example including a Covid-19 related example is in notebooks/covid.ipynb:

1.2.1 Operator Example

Here is an example from the RCSB PDB Search API page, using the operator syntax. This query finds symmetric dimers having a twofold rotation with the DNA-binding domain of a heat-shock transcription factor.

```
from rcsbsearchapi.search import TextQuery
from rcsbsearchapi import rcsb_attributes as attrs

# Create terminals for each query
q1 = TextQuery("heat-shock transcription factor")
q2 = attrs.rcsb_struct_symmetry.symbol == "C2"
q3 = attrs.rcsb_struct_symmetry.kind == "Global Symmetry"
q4 = attrs.rcsb_entry_info.polymer_entity_count_DNA >= 1

# combined using bitwise operators (&, |, ~, etc)
query = q1 & (q2 & q3 & q4)

# Call the query to execute it
for assemblyid in query("assembly"):
    print(assemblyid)
```

For a full list of attributes, please refer to the RCSB PDB schema.

1.2.2 Fluent Example

Here is the same example using the fluent syntax

```
from rcsbsearchapi.search import TextQuery, AttributeQuery, Attr

# Start with a Attr or TextQuery, then add terms
results = TextQuery("heat-shock transcription factor").and_(
    # Add attribute node as fully-formed AttributeQuery
    AttributeQuery(attribute="rcsb_struct_symmetry.symbol", operator="exact_match",␣
↪value="C2") \
    # Add attribute node as Attr with chained operations
    .and_(Attr("rcsb_struct_symmetry.kind")).exact_match("Global Symmetry") \
    # Add attribute node by name (converted to Attr) with chained operations
    .and_("rcsb_entry_info.polymer_entity_count_DNA").greater_or_equal(1)
    ).exec("assembly")

# Exec produces an iterator of IDs
for assemblyid in results:
    print(assemblyid)
```

1.2.3 Structural Attribute Search and Chemical Attribute Search Combination

Grouping of a Structural Attribute query and Chemical Attribute query is permitted as long as grouping is done correctly and search services are specified accordingly. More details on attributes that are available for text searches can be found on the RCSB PDB Search API page.

```
from rcsbsearchapi.const import CHEMICAL_ATTRIBUTE_SEARCH_SERVICE, STRUCTURE_
↪ATTRIBUTE_SEARCH_SERVICE
from rcsbsearchapi.search import AttributeQuery

# By default, service is set to "text" for structural attribute search
q1 = AttributeQuery("exptl.method", "exact_match", "electron microscopy",
                    STRUCTURE_ATTRIBUTE_SEARCH_SERVICE # this constant specifies "text
↪" service
                    )

# Need to specify chemical attribute search service - "text_chem"
q2 = AttributeQuery("drugbank_info.brand_names", "contains_phrase", "tylenol",
                    CHEMICAL_ATTRIBUTE_SEARCH_SERVICE # this constant specifies "text_
↪chem" service
                    )

query = q1 & q2 # combining queries

list(query())
```

1.2.4 Computed Structure Models

The RCSB PDB Search API page provides information on how to include Computed Structure Models (CSMs) into a search query. Here is a code example below. This query returns IDs for experimental and computed structure models associated with "hemoglobin". Queries for *only* computed models or *only* experimental models can also be made (default).

```
from rcsbsearchapi.search import TextQuery

q1 = TextQuery("hemoglobin")

# add parameter as a list with either "computational" or "experimental" or both as␣
↪list values
q2 = q1(return_content_type=["computational", "experimental"])

list(q2)
```

1.2.5 Return Types and Attribute Search

A search query can return different result types when a return type is specified. Below are examples on specifying return types Polymer Entities, Non-polymer Entities, Polymer Instances, and Molecular Definitions, using a Structure Attribute query. More information on return types can be found in the RCSB PDB Search API page.

```
from rcsbsearchapi.search import AttributeQuery

q1 = AttributeQuery("rcsb_entry_container_identifiers.entry_id", "in", ["4HHB"]) #␣
↪query for 4HHB deoxyhemoglobin

# Polymer entities
for poly in q1("polymer_entity"): # include return type as a string parameter for␣
↪query object
    print(poly)

# Non-polymer entities
for nonPoly in q1("non_polymer_entity"):
    print(nonPoly)

# Polymer instances
for polyInst in q1("polymer_instance"):
    print(polyInst)

# Molecular definitions
for mol in q1("mol_definition"):
    print(mol)
```

1.2.6 Counting Results

If only the number of results is desired, the count function can be used. This query returns the number of experimental models associated with "hemoglobin".

```
from rcsbsearchapi.search import TextQuery

q1 = TextQuery("hemoglobin")

# N.B., Just as shown above for `query()`, `return_type` and `return_content_type` can␣
↪also be specified as parameters to `count()`
q1.count()
```

1.2.7 Obtaining Scores for Results

Results can be returned alongside additional metadata, including result scores. To return this metadata, set the `results_verbosity` parameter to "verbose" (all metadata), "minimal" (scores only), or "compact" (default, no metadata). If set to "verbose" or "minimal", results will be returned as a list of dictionaries. For example, here we get all experimental models associated with "hemoglobin", along with their scores.

```
from rcsbsearchapi.search import TextQuery

q1 = TextQuery("hemoglobin")
for idscore in list(q1(results_verbosity="minimal")):
    print(idscore)
```

1.2.8 Protein Sequence Search Example

Below is an example from the RCSB PDB Search API page, using the sequence search function. This query finds macromolecular PDB entities that share 90% sequence identity with GTPase HRas protein from *Gallus gallus* (*Chicken*).

```
from rcsbsearchapi.search import SequenceQuery

# Use SequenceQuery class and add parameters
results = SequenceQuery("MTEYKLVVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGET" +
                        "CLLDILDTAGQEEYSAMRDQYMRTGEGFLCVFAINNTKSFEDIHQYREQI" +
                        "KRVKDSDDVPMVLVGNKCDLPARTVETRQAQDLARSYGIPYIETSAKTRQ" +
                        "GVEDAFYTLVREIRQHKLRKLNPPDESGPGCMNCKCVIS", 1, 0.9)

# results("polymer_entity") produces an iterator of IDs with return type - polymer␣
↪entities
for polyid in results("polymer_entity"):
    print(polyid)
```

1.2.9 Sequence Motif Search Example

Below is an example from the RCSB PDB Search API page, using the sequence motif search function. This query retrives occurences of the His2/Cys2 Zinc Finger DNA-binding domain as represented by its PROSITE signature.

```
from rcsbsearchapi.search import SeqMotifQuery

# Use SeqMotifQuery class and add parameters
results = SeqMotifQuery("C-x(2,4)-C-x(3)-[LIVMFYWC]-x(8)-H-x(3,5)-H.",
                        pattern_type="prosite",
                        sequence_type="protein")

# results("polymer_entity") produces an iterator of IDs with return type - polymer␣
↪entities
for polyid in results("polymer_entity"):
    print(polyid)
```

You can also use a regular expression (RegEx) to make a sequence motif search. As an example, here is a query for the zinc finger motif that binds Zn in a DNA-binding domain:

```
from rcsbsearchapi.search import SeqMotifQuery

results = SeqMotifQuery("C.{2,4}C.{12}H.{3,5}H", pattern_type="regex", sequence_type=
↪"protein")

for polyid in results("polymer_entity"):
    print(polyid)
```

You can use a standard amino acid sequence to make a sequence motif search. X can be used to allow any amino acid in that position. As an example, here is a query for SH3 domains:

```
from rcsbsearchapi.search import SeqMotifQuery

# By default, the pattern_type argument is "simple" and the sequence_type argument is
↪"protein".
results = SeqMotifQuery("XPPXP")  # X is used as a "variable residue" and can be any␣
↪amino acid.

for polyid in results("polymer_entity"):
    print(polyid)
```

All 3 of these pattern types can be used to search for DNA and RNA sequences as well. Demonstrated are 2 queries, one DNA and one RNA, using the simple pattern type:

```
from rcsbsearchapi.search import SeqMotifQuery

# DNA query: this is a query for a T-Box.
dna = SeqMotifQuery("TCACACCT", sequence_type="dna")

print("DNA results:")
for polyid in dna("polymer_entity"):
    print(polyid)

# RNA query: 6C RNA motif
rna = SeqMotifQuery("CCCCCC", sequence_type="rna")
print("RNA results:")
for polyid in rna("polymer_entity"):
    print(polyid)
```

1.2.10 Structure Similarity Query Example

The PDB archive can be queried using the 3D shape of a protein structure. To perform this query, 3D protein structure data must be provided as an input or parameter, A chain ID or assembly ID must be specified, whether the input structure data should be compared to Assemblies or Polymer Entity Instance (Chains) is required, and defining the search type as either strict or relaxed is required. More information on how Structure Similarity Queries work can be found on the RCSB PDB Structure Similarity Search page.

```
from rcsbsearchapi.search import StructSimilarityQuery

# Basic query: querying using entry ID and default values assembly ID "1", operator
↪"strict", and target search space "Assemblies"
q1 = StructSimilarityQuery(entry_id="4HHB")

# Same example but with parameters explicitly specified
q1 = StructSimilarityQuery(structure_search_type="entry_id",
                           entry_id="4HHB",
                           structure_input_type="assembly_id",
                           assembly_id="1",
                           operator="strict_shape_match",
                           target_search_space="assembly"
                           )
for id in q1("assembly"):
    print(id)
```

Below is a more complex example that utilizes chain ID, relaxed search operator, and polymer entity instance or target search space. Specifying whether the input structure type is chain id or assembly id is very important. For example, specifying chain ID as the input structure type but inputting an assembly ID can lead to an error.

```
from rcsbsearchapi.search import StructSimilarityQuery

# More complex query with entry ID value "4HHB", chain ID "B", operator "relaxed",␣
↪and target search space "Chains"
q2 = StructSimilarityQuery(structure_search_type="entry_id",
                                  entry_id="4HHB",
                                  structure_input_type="chain_id",
                                  chain_id="B",
                                  operator="relaxed_shape_match",
                                  target_search_space="polymer_entity_instance")
list(q2())
```

Structure similarity queries also allow users to upload a file from their local computer or input a file url from the website to query the PDB archive for similar proteins. The file represents a target protein structure in the file formats “cif”, “bcif”, “pdb”, “cif.gz”, or “pdb.gz”. If a user wants to use a file url for queries, the user must specify the structure search type, the value (being the url), and the file format of the file. This is also the same case for file upload, except the value is the absolute path leading to the file that is in the local machine. An example for file url is below for 4HHB (hemoglobin).

```
from rcsbsearchapi.search import StructSimilarityQuery

q3 = StructSimilarityQuery(structure_search_type="file_url",
                           file_url="https://files.rcsb.org/view/4HHB.cif",
                           file_format="cif")
list(q3())

# If you want to upload your own structure file for similarity search, you can do so␣
↪by using the `file_path` parameter:
```

(continues on next page)

(continued from previous page)

```
q4 = StructSimilarityQuery(structure_search_type="file_upload",
                           file_path="/PATH/TO/FILE.cif",  # specify local model file␣
↪path
                           file_format="cif")
list(q4())
```

1.2.11 Structure Motif Query Examples

The PDB Archive can also be queried by using a "motif" found in these 3D structures. To perform this type of query, an entry_id or a file URL/path must be provided, along with residues (which are parts of 3D structures.) This is the bare minimum needed to make a search, but there are lots of other parameters that can be added to a Structure Motif Query (see full search schema).

To make a Structure Motif Query, you must first define anywhere from 2-10 "residues" that will be used in the query. Each individual residue has a Chain ID, Operator, Residue Number, and Exchanges (optional) that can be declared in that order using positonal arguments, or using the "chain_id", "struct_oper_id", and "label_seq_id" to define what parameter you are passing through. All 3 of the required parameters must be included, or the package will throw an AssertionError.

Each residue can only have a maximum of 4 Exchanges, and each query can only have 16 exchanges total. Violating any of these rules will cause the package to throw an AssertionError.

Examples of how to instantiate Residues can be found below. These can then be put into a list and passed through to a Structure Motif Query.

```
from rcsbsearchapi.search import StructureMotifResidue

# construct a Residue with a Chain ID of A, an operator of 1, a residue
# number of 192, and Exchanges of "LYS" and "HIS"
Res1 = StructureMotifResidue("A", "1", 192, ["LYS", "HIS"])
# as for what is a valid "Exchange", the package provides these as a literal,
# and they should be type checked.

# you can also specify the arguments:
# this query is the same as above.
Res2 = StructureMotifResidue(struct_oper_id="1", chain_id="A", exchanges=["LYS", "HIS
↪"], label_seq_id=192)

# after delcaring a minimum of 2 and as many as 10 residues, they can be passed into␣
↪a list for use in the query itself:
Res3 = StructureMotifResidue("A", "1", 162)  # exchanges are optional

ResList = [Res1, Res3]
```

From there, these Residues can be used in a query. As stated before, you can only include 2 - 10 residues in a query. If you fail to provide residues for a query, or provide the wrong amount, the package will throw a ValueError.

For a Structure Motif Query using an entry_id, the only other necessary value that must be passed into the query is the residue list. The default type of query is an entry_id query.

As this type of query has a lot of optional parameters, do *not* use positional arguments as more than likely an error will occur.

Below is an example of a basic entry_id Structure Motif Query, with the residues declared earlier:

```
from rcsbsearchapi.search import StructMotifQuery

q1 = StructMotifQuery(entry_id="2MNR", residue_ids=ResList)
list(q1())
```

Like with Structure Similarity Queries, a file url or filepath can also be provided to the program. These can take the place of an entry_id.

For a file url query, you *must* provide both a valid file URL (a string), and the file's file extension (also as a string). Failure to provide these elements correctly will cause the package to throw an AssertionError.

Below is an example of the same query as above, only this time providing a file url:

```
link = "https://files.rcsb.org/view/2MNR.cif"
q2 = StructMotifQuery(structure_search_type="file_url", url=link, file_extension="cif
↪", residue_ids=ResList)
# structure_search_type MUST be provided. A mismatched query type will cause an error.
↪
list(q2())
```

Like with Structure Similarity Queries, a filepath to a file may also be provided. This file must be a valid file accepted by the search API. A file extension must also be provided with the file upload.

The query would look something like this:

```
filepath = "/absolute/path/to/file.cif"
q3 = StructMotifQuery(structure_search_type="file_upload", file_path=filepath, file_
↪extension="cif", residue_ids=ResList)

list(q3())
```

There are many additional parameters that Structure Motif Query supports. These include a variety of features such as backbone distance tolerance, side chain distance tolerance, angle tolerance, RMSD cutoff, limits (stop searching after this many hits), atom pairing schemes, motif pruning strategy, allowed structures, and excluded structures. These can be mixed and matched as needed to make accurate and useful queries. All of these have some default value which is used when a parameter isn't provided. These parameters conform to the defaults used by the Search API.

Below will demonstrate how to define these parameters using non-positional arguments:

```
# specifying backbone distance tolerance: 0-3, default is 1
# allowed backbone distance tolerance in Angstrom.
backbone = StructMotifQuery(entry_id="2MNR", backbone_distance_tolerance=2, residue_
↪ids=ResList)
list(backbone())

# specifying sidechain distance tolerance: 0-3, default is 1
# allowed side-chain distance tolerance in Angstrom.
sidechain = StructMotifQuery(entry_id="2MNR", side_chain_distance_tolerance=2,␣
↪residue_ids=ResList)
list(sidechain())

# specifying angle tolerance: 0-3, default is 1
# allowed angle tolerance in multiples of 20 degrees.
angle = StructMotifQuery(entry_id="2MNR", angle_tolerance=2, residue_ids=ResList)
list(angle())

# specifying RMSD cutoff: >=0, default is 2
# Threshold above which hits will be filtered by RMSD
```

(continues on next page)

(continued from previous page)

```
rmsd = StructMotifQuery(entry_id="2MNR", rmsd_cutoff=1, residue_ids=ResList)
list(rmsd())

# specifying limit: >=0, default excluded
# Stop accepting results after this many hits.
limit = StructMotifQuery(entry_id="2MNR", limit=100, residue_ids=ResList)
list(limit())

# specifying atom pairing scheme, default = "SIDE_CHAIN"
# ENUM: "ALL", "BACKBONE", "SIDE_CHAIN", "PSUEDO_ATOMS"
# this is typechecked by a literal.
# Which atoms to consider to compute RMSD scores and transformations.
atom = StructMotifQuery(entry_id="2MNR", atom_pairing_scheme="ALL", residue_
↪ids=ResList)
list(atom())

# specifying motif pruning strategy, default = "KRUSKAL"
# ENUM: "NONE", "KRUSKAL"
# this is typechecked by a literal in the package.
# Specifies how many query motifs are "pruned". KRUSKAL leads to less stringent␣
↪queries, and faster results.
pruning = StructMotifQuery(entry_id="2MNR", motif_pruning_strategy="NONE", residue_
↪ids=ResList)
list(pruning())

# specifying allowed structures, default excluded
# specify the structures you wish to allow in the return result. As an example,
# we could only allow the results from the limited query we ran earlier.
allowed = StructMotifQuery(entry_id="2MNR", allowed_structures=list(limit()), residue_
↪ids=ResList)
list(allowed())

# specifying structures to exclude, default excluded
# specify structures to exclude from a query. We could, for example,
# exclude the results of the previous allowed query.
excluded = StructMotifQuery(entry_id="2MNR", excluded_structures=list(allowed()),␣
↪residue_ids=ResList)
list(excluded())
```

The Structure Motif Query can be used to make some very specific queries. Below is an example of a query that retrives occurances of the enolase superfamily, a group of proteins diverse in sequence and structure that are all capable of abstracting a proton from a carboxylic acid. Position-specific exchanges are crucial to represent this superfamily accurately.

```
Res1 = StructureMotifResidue("A", "1", 162, ["LYS", "HIS"])
Res2 = StructureMotifResidue("A", "1", 193)
Res3 = StructureMotifResidue("A", "1", 219)
Res4 = StructureMotifResidue("A", "1", 245, ["GLU", "ASP", "ASN"])
Res5 = StructureMotifResidue("A", "1", 295, ["HIS", "LYS"])

ResList = [Res1, Res2, Res3, Res4, Res5]

query = StructMotifQuery(entry_id="2MNR", residue_ids=ResList)

list(query())
```

1.2.12 Chemical Similarity Query

When you have unique chemical information (e.g., a chemical formula or descriptor) you can use this information to find chemical components (e.g., drugs, inhibitors, modified residues, or building blocks such as amino acids, nucleotides, or sugars), so that it is similar to the formula or descriptor used in the query (perhaps one or two atoms/groups are different), is part of a larger molecule (i.e., the specified formula/descriptor is a substructure), or is exactly or very closely matches the formula or descriptor used in the query.

The search can also be used to identify PDB structures that include the chemical component(s) which match or are similar to the query. These structures can then be examined to learn about the interactions of the component within the structure. More information on Chemical Similarity Queries can be found on the RCSB PDB Chemical Similarity Search page.

To do a Chemical Similarity query, you must first specify one of two possible query options which are formula and descriptors. Formula allows queries to be made by providing a chemical formula. Descriptors allow you to search by chemical notations for example. Each Query option has its own distinct set of parameters, but both options require a value.

The formula query option comes with a match subset parameter which allows users to search chemical components whose formula exactly match the query or matches any portion of the query. The descriptor query option comes with a descriptor type parameter and match type parameter. The descriptor type parameter specifies what type of descriptor the input value is. There are two options which are SMILES (Simplified Molecular Input Line Entry Specification) and InChI (International Chemical Identifier). The match type parameter has six options which are Similar Ligands (Quick Screen), Similar Ligands (Stereospecific), Similar Ligands (including Stereoisomers), Substructure (Stereospecific), Substructure (including Stereoisomers), and Exact match.

When doing Chemical Similarity Queries in this tool, it is important to note that by default the query option is set to formula and match subset is set to False. An example of how that looks like is below.

```
from rcsbsearchapi.search import ChemSimilarityQuery

# Basic query with default values: query type = formula and match subset = False
q1 = ChemSimilarityQuery(value="C12 H17 N4 O S")

# Same example but with all the parameters listed
q1 = ChemSimilarityQuery(value="C12 H17 N4 O S",
                         query_type="formula",
                         match_subset=False)
list(q1())
```

Below is are two examples of using query option descriptor. Both descriptor type parameters are also used.

```
from rcsbsearchapi.search import ChemSimilarityQuery

# Query with type = descriptor, descriptor type = SMILES, match type = similar␣
↪ligands (sterospecific) or graph-relaxed-stereo
q2 = ChemSimilarityQuery(value="Cc1c(sc[n+]1Cc2cnc(nc2N)C)CCO",
                         query_type="descriptor",
                         descriptor_type="SMILES",
                         match_type="graph-relaxed-stereo")
list(q2())
```

```
from rcsbsearchapi.search import ChemSimilarityQuery

# Query with type = descriptor, descriptor type = InChI, match type = substructure␣
↪(sterospecific) or sub-struct-graph-relaxed-stereo
q3 = ChemSimilarityQuery(value="InChI=1S/C13H10N2O4/c16-10-6-5-9(11(17)14-10)15-
↪12(18)7-3-1-2-4-8(7)13(15)19/h1-4,9H,5-6H2,(H,14,16,17)/t9-/m0/s1",
```

(continues on next page)

(continued from previous page)

```
                          query_type="descriptor",
                          descriptor_type="InChI",
                          match_type="sub-struct-graph-relaxed-stereo")
list(q3())
```

1.2.13 Faceted Query Examples

In order to group and perform calculations and statistics on PDB data by using a simple search query, you can use a faceted query (or facets). Facets arrange search results into categories (buckets) based on the requested field values. More information on Faceted Queries can be found here. All facets should be provided with `name`, `aggregation_type`, and `attribute` values. Depending on the aggregation type, other parameters must also be specified. The `facets()` function runs the query `q` using the specified facet(s), and returns a list of dictionaries:

```
from rcsbsearchapi.search import AttributeQuery, Facet, Range

q = AttributeQuery("rcsb_accession_info.initial_release_date", operator="greater",␣
↪value="2019-08-20")
q.facets(facets=Facet(name="Methods", aggregation_type="terms", attribute="exptl.
↪method"))
```

Term Facets

Terms faceting is a multi-bucket aggregation where buckets are dynamically built - one per unique value. We can specify the minimum count (`>= 0`) for a bucket to be returned using the parameter `min_interval_population` (default value `1`). We can also control the number of buckets returned (`<= 65336`) using the parameter `max_num_intervals` (default value `65336`).

```
# This is the default query, used by the RCSB Search API when no query is explicitly␣
↪specified.
# This default query will be used for most of the examples found below for faceted␣
↪queries.
base_q = AttributeQuery("rcsb_entry_info.structure_determination_methodology",␣
↪operator="exact_match", value="experimental")

base_q.facets(facets=Facet(name="Journals", aggregation_type="terms", attribute="rcsb_
↪primary_citation.rcsb_journal_abbrev", min_interval_population=1000))
```

Histogram Facets

Histogram facets build fixed-sized buckets (intervals) over numeric values. The size of the intervals must be specified in the parameter `interval`. We can also specify `min_interval_population` if desired.

```
base_q.facets(return_type="polymer_entity", facets=Facet(name="Formula Weight",␣
↪aggregation_type="histogram", attribute="rcsb_polymer_entity.formula_weight",␣
↪interval=50, min_interval_population=1))
```

Date Histogram Facets

Similar to histogram facets, date histogram facetes build buckets over date values. For date histogram aggregations, we must specify `interval="year"`. Again, we may also specify `min_interval_population`.

```
base_q.facets(facets=Facet(name="Release Date", aggregation_type="date_histogram",␣
↪attribute="rcsb_accession_info.initial_release_date", interval="year", min_interval_
↪population=1))
```

Range Facets

We can define the buckets ourselves by using range facets. In order to specify the ranges, we use the `Range` class. Note that the range includes the `start` value and excludes the `end` value (`include_lower` and `include_upper` should not be specified). If the `start` or `end` is omitted, the minimum or maximum boundaries will be used by default. The buckets should be provided as a list of `Range` objects to the `ranges` parameter.

```
base_q.facets(facets=Facet(name="Resolution Combined", aggregation_type="range",␣
↪attribute="rcsb_entry_info.resolution_combined", ranges=[Range(start=None,end=2),␣
↪Range(start=2, end=2.2), Range(start=2.2, end=2.4), Range(start=4.6, end=None)]))
```

Date Range Facets

Date range facets allow us to specify date values as bucket ranges, using date math expressions.

```
base_q.facets(facets=Facet(name="Release Date", aggregation_type="date_range",␣
↪attribute="rcsb_accession_info.initial_release_date", ranges=[Range(start=None,end=
↪"2020-06-01||-12M"), Range(start="2020-06-01", end="2020-06-01||+12M"), Range(start=
↪"2020-06-01||+12M", end=None)]))
```

Cardinality Facets

Cardinality facets return a single value: the count of distinct values returned for a given field. A `precision_threshold` (`<= 40000`, default value `40000`) may be specified.

```
base_q.facets(facets=Facet(name="Organism Names Count", aggregation_type="cardinality
↪", attribute="rcsb_entity_source_organism.ncbi_scientific_name"))
```

Multidimensional Facets

Complex, multi-dimensional aggregations are possible by specifying additional facets in the `nested_facets` parameter, as in the example below:

```
f1 = Facet(name="Polymer Entity Types", aggregation_type="terms", attribute="rcsb_
↪entry_info.selected_polymer_entity_types")
f2 = Facet(name="Release Date", aggregation_type="date_histogram", attribute="rcsb_
↪accession_info.initial_release_date", interval="year")
base_q.facets(facets=Facet(name="Experimental Method", aggregation_type="terms",␣
↪attribute="rcsb_entry_info.experimental_method", nested_facets=[f1, f2]))
```

Filter Facets

Filters allow us to filter documents that contribute to bucket count. Similar to queries, we can group several `TerminalFilters` into a single `GroupFilter`. We can combine a filter with a facet using the `FilterFacet` class. Terminal filters should specify an `attribute` and `operator`, as well as possible a `value` and whether or not it should be a `negation` and/or `case_sensitive`. Group filters should specify a `logical_operator` (which should be either `"and"` or `"or"`) and a list of filters (`nodes`) that should be combined. Finally, the `FilterFacet` should be provided with a filter and a (list of) facet(s). Here are some examples:

```
from rcsbsearchapi.search import TerminalFilter, GroupFilter, FilterFacet
tf1 = TerminalFilter(attribute="rcsb_polymer_instance_annotation.type", operator=
↪"exact_match", value="CATH")
tf2 = TerminalFilter(attribute="rcsb_polymer_instance_annotation.annotation_lineage.id
↪", operator="in", value=["2.140.10.30", "2.120.10.80"])
ff2 = FilterFacet(filters=tf2, facets=Facet("CATH Domains", "terms", "rcsb_polymer_
↪instance_annotation.annotation_lineage.id", min_interval_population=1))
ff1 = FilterFacet(filters=tf1, facets=ff2)
base_q.facets("polymer_instance", ff1)

tf1 = TerminalFilter(attribute="rcsb_struct_symmetry.kind", operator="exact_match",␣
↪value="Global Symmetry", negation=False)
f2 = Facet(name="ec_terms", aggregation_type="terms", attribute="rcsb_polymer_entity.
↪rcsb_ec_lineage.id")
f1 = Facet(name="sym_symbol_terms", aggregation_type="terms", attribute="rcsb_struct_
↪symmetry.symbol", nested_facets=f2)
ff = FilterFacet(filters=tf1, facets=f1)
q1 = AttributeQuery("rcsb_assembly_info.polymer_entity_count", operator="equals",␣
↪value=1)
q2 = AttributeQuery("rcsb_assembly_info.polymer_entity_instance_count", operator=
↪"greater", value=1)
q = q1 & q2
q.facets("assembly", ff)

tf1 = TerminalFilter(attribute="rcsb_polymer_entity_group_membership.aggregation_
↪method", operator="exact_match", value="sequence_identity")
tf2 = TerminalFilter(attribute="rcsb_polymer_entity_group_membership.similarity_cutoff
↪", operator="equals", value=100)
gf = GroupFilter(logical_operator="and", nodes=[tf1, tf2])
ff = FilterFacet(filters=gf, facets=Facet("Distinct Protein Sequence Count",
↪"cardinality", "rcsb_polymer_entity_group_membership.group_id"))
base_q.facets("polymer_entity", ff)
```

CHAPTER

TWO

QUERIES

Two syntaxes are available for constructing queries: an "operator" API using python's comparators, and a "fluent" API where terms are chained together. Which to use is a matter of preference, and both construct the same query object.

2.1 Operator syntax

Searches are built up from a series of `Terminal` nodes, which compare structural attributes to some search value. In the operator syntax, python's comparator operators are used to construct the comparison. The operators are overloaded to return `Terminal` objects for the comparisons.

```
from rcsbsearchapi.search import TextQuery
from rcsbsearchapi import rcsb_attributes as attrs

# Create terminals for each query
q1 = TextQuery("heat-shock transcription factor")
q2 = attrs.rcsb_struct_symmetry.symbol == "C2"
q3 = attrs.rcsb_struct_symmetry.kind == "Global Symmetry"
q4 = attrs.rcsb_entry_info.polymer_entity_count_DNA >= 1
```

Attributes are available from the rcsb_attributes object and can be tab-completed. They can additionally be constructed from strings using the `Attr(attribute)` constructor. For a full list of attributes, please refer to the RCSB PDB schema.

`Terminals` are combined into `Groups` using python's bitwise operators. This is analogous to how bitwise operators act on python `set` objects. The operators are lazy and won't perform the search until the query is executed.

```
query = q1 & (q2 & q3 & q4)  # AND of all queries
```

AND (`&`), OR (`|`), and terminal negation (`~`) are implemented directly by the API, but the python package also implements set difference (`-`), symmetric difference (`^`), and general negation by transforming the query.

Queries are executed by calling them as functions. They return an iterator of result identifiers.

```
results = set(query())
```

By default, the query will return "entry" results (PDB IDs). It is also possible to query other types of results (see return-types for options):

```
assemblies = set(query("assembly"))
```

2.1.1 Fluent syntax

The operator syntax is great for simple queries, but requires parentheses or temporary variables for complex nested queries. In these cases the fluent syntax may be clearer. Queries are built up by appending operations sequentially.

```
from rcsbsearchapi.search import TextQuery, AttributeQuery, Attr

# Start with a Attr or TextQuery, then add terms
results = TextQuery("heat-shock transcription factor").and_(
    # Add attribute node as fully-formed AttributeQuery
    AttributeQuery(attribute="rcsb_struct_symmetry.symbol", operator="exact_match",␣
↪value="C2") \
    # Add attribute node as Attr with chained operations
    .and_(Attr("rcsb_struct_symmetry.kind")).exact_match("Global Symmetry") \
    # Add attribute node by name (converted to Attr) with chained operations
    .and_("rcsb_entry_info.polymer_entity_count_DNA").greater_or_equal(1)
    ).exec("assembly")

# Exec produces an iterator of IDs
for assemblyid in results:
    print(assemblyid)
```

2.1.2 Structural Attribute Search and Chemical Attribute Search Combination

Grouping of a Structural Attribute query and Chemical Attribute query is permitted as long as grouping is done correctly and search services are specified accordingly. More details on attributes that are available for text searches can be found on the RCSB PDB Search API page.

```
from rcsbsearchapi.const import CHEMICAL_ATTRIBUTE_SEARCH_SERVICE, STRUCTURE_
↪ATTRIBUTE_SEARCH_SERVICE
from rcsbsearchapi.search import AttributeQuery

# By default, service is set to "text" for structural attribute search
q1 = AttributeQuery("exptl.method", "exact_match", "electron microscopy",
                    STRUCTURE_ATTRIBUTE_SEARCH_SERVICE # this constant specifies "text
↪" service
                    )

# Need to specify chemical attribute search service - "text_chem"
q2 = AttributeQuery("drugbank_info.brand_names", "contains_phrase", "tylenol",
                    CHEMICAL_ATTRIBUTE_SEARCH_SERVICE # this constant specifies "text_
↪chem" service
                    )

query = q1 & q2 # combining queries

list(query())
```

2.1.3 Computed Structure Models

The RCSB PDB Search API page provides information on how to include Computed Structure Models (CSMs) into a search query. Here is a code example below. This query returns IDs for experimental and computed structure models associated with "hemoglobin". Queries for *only* computed models or *only* experimental models can also be made (default).

```
from rcsbsearchapi.search import TextQuery

q1 = TextQuery("hemoglobin")

# add parameter as a list with either "computational" or "experimental" or both as␣
↪list values
q2 = q1(return_content_type=["computational", "experimental"])

list(q2)
```

2.1.4 Return Types and Attribute Search

A search query can return different result types when a return type is specified. Below are examples on specifying return types Polymer Entities, Non-polymer Entities, Polymer Instances, and Molecular Definitions, using a Structure Attribute query. More information on return types can be found in the RCSB PDB Search API page.

```
from rcsbsearchapi.search import AttributeQuery

q1 = AttributeQuery("rcsb_entry_container_identifiers.entry_id", "in", ["4HHB"]) #␣
↪query for 4HHB deoxyhemoglobin

# Polymer entities
for poly in q1("polymer_entity"): # include return type as a string parameter for␣
↪query object
    print(poly)

# Non-polymer entities
for nonPoly in q1("non_polymer_entity"):
    print(nonPoly)

# Polymer instances
for polyInst in q1("polymer_instance"):
    print(polyInst)

# Molecular definitions
for mol in q1("mol_definition"):
    print(mol)
```

2.1.5 Counting Results

If only the number of results is desired, the count function can be used. This query returns the number of experimental models associated with "hemoglobin".

```
from rcsbsearchapi.search import TextQuery

q1 = TextQuery("hemoglobin")

# N.B., Just as shown above for `query()`, `return_type` and `return_content_type` can␣
↪also be specified as parameters to `count()`
q1.count()
```

2.1.6 Obtaining Scores for Results

Results can be returned alongside additional metadata, including result scores. To return this metadata, set the `results_verbosity` parameter to "verbose" (all metadata), "minimal" (scores only), or "compact" (default, no metadata). If set to "verbose" or "minimal", results will be returned as a list of dictionaries. For example, here we get all experimental models associated with "hemoglobin", along with their scores.

```
from rcsbsearchapi.search import TextQuery

q1 = TextQuery("hemoglobin")
for idscore in list(q1(results_verbosity="minimal")):
    print(idscore)
```

2.1.7 Protein Sequence Search Example

Below is an example from the RCSB PDB Search API page, using the sequence search function. This query finds macromolecular PDB entities that share 90% sequence identity with GTPase HRas protein from *Gallus gallus* (*Chicken*).

```
from rcsbsearchapi.search import SequenceQuery

# Use SequenceQuery class and add parameters
results = SequenceQuery("MTEYKLVVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGET" +
                        "CLLDILDTAGQEEYSAMRDQYMRTGEGFLCVFAINNTKSFEDIHQYREQI" +
                        "KRVKDSDDVPMVLVGNKCDLPARTVETRQAQDLARSYGIPYIETSAKTRQ" +
                        "GVEDAFYTLVREIRQHKLRKLNPPDESGPGCMNCKCVIS", 1, 0.9)

# results("polymer_entity") produces an iterator of IDs with return type - polymer␣
↪entities
for polyid in results("polymer_entity"):
    print(polyid)
```

2.1.8 Sequence Motif Search Example

Below is an example from the RCSB PDB Search API page, using the sequence motif search function. This query retrives occurences of the His2/Cys2 Zinc Finger DNA-binding domain as represented by its PROSITE signature.

```
from rcsbsearchapi.search import SeqMotifQuery

# Use SeqMotifQuery class and add parameters
results = SeqMotifQuery("C-x(2,4)-C-x(3)-[LIVMFYWC]-x(8)-H-x(3,5)-H.",
                        pattern_type="prosite",
                        sequence_type="protein")

# results("polymer_entity") produces an iterator of IDs with return type - polymer␣
↪entities
for polyid in results("polymer_entity"):
    print(polyid)
```

You can also use a regular expression (RegEx) to make a sequence motif search. As an example, here is a query for the zinc finger motif that binds Zn in a DNA-binding domain:

```
from rcsbsearchapi.search import SeqMotifQuery

results = SeqMotifQuery("C.{2,4}C.{12}H.{3,5}H", pattern_type="regex", sequence_type=
↪"protein")

for polyid in results("polymer_entity"):
    print(polyid)
```

You can use a standard amino acid sequence to make a sequence motif search. X can be used to allow any amino acid in that position. As an example, here is a query for SH3 domains:

```
from rcsbsearchapi.search import SeqMotifQuery

# By default, the pattern_type argument is "simple" and the sequence_type argument is
↪"protein".
results = SeqMotifQuery("XPPXP")  # X is used as a "variable residue" and can be any␣
↪amino acid.

for polyid in results("polymer_entity"):
    print(polyid)
```

All 3 of these pattern types can be used to search for DNA and RNA sequences as well. Demonstrated are 2 queries, one DNA and one RNA, using the simple pattern type:

```
from rcsbsearchapi.search import SeqMotifQuery

# DNA query: this is a query for a T-Box.
dna = SeqMotifQuery("TCACACCT", sequence_type="dna")

print("DNA results:")
for polyid in dna("polymer_entity"):
    print(polyid)

# RNA query: 6C RNA motif
rna = SeqMotifQuery("CCCCCC", sequence_type="rna")
print("RNA results:")
for polyid in rna("polymer_entity"):
    print(polyid)
```

2.1.9 Structure Similarity Query Example

The PDB archive can be queried using the 3D shape of a protein structure. To perform this query, 3D protein structure data must be provided as an input or parameter, A chain ID or assembly ID must be specified, whether the input structure data should be compared to Assemblies or Polymer Entity Instance (Chains) is required, and defining the search type as either strict or relaxed is required. More information on how Structure Similarity Queries work can be found on the RCSB PDB Structure Similarity Search page.

```
from rcsbsearchapi.search import StructSimilarityQuery

# Basic query: querying using entry ID and default values assembly ID "1", operator
↪"strict", and target search space "Assemblies"
q1 = StructSimilarityQuery(entry_id="4HHB")

# Same example but with parameters explicitly specified
q1 = StructSimilarityQuery(structure_search_type="entry_id",
                           entry_id="4HHB",
                           structure_input_type="assembly_id",
                           assembly_id="1",
                           operator="strict_shape_match",
                           target_search_space="assembly"
                           )
for id in q1("assembly"):
    print(id)
```

Below is a more complex example that utilizes chain ID, relaxed search operator, and polymer entity instance or target search space. Specifying whether the input structure type is chain id or assembly id is very important. For example, specifying chain ID as the input structure type but inputting an assembly ID can lead to an error.

```
from rcsbsearchapi.search import StructSimilarityQuery

# More complex query with entry ID value "4HHB", chain ID "B", operator "relaxed",␣
↪and target search space "Chains"
q2 = StructSimilarityQuery(structure_search_type="entry_id",
                                  entry_id="4HHB",
                                  structure_input_type="chain_id",
                                  chain_id="B",
                                  operator="relaxed_shape_match",
                                  target_search_space="polymer_entity_instance")
list(q2())
```

Structure similarity queries also allow users to upload a file from their local computer or input a file url from the website to query the PDB archive for similar proteins. The file represents a target protein structure in the file formats “cif”, “bcif”, “pdb”, “cif.gz”, or “pdb.gz”. If a user wants to use a file url for queries, the user must specify the structure search type, the value (being the url), and the file format of the file. This is also the same case for file upload, except the value is the absolute path leading to the file that is in the local machine. An example for file url is below for 4HHB (hemoglobin).

```
from rcsbsearchapi.search import StructSimilarityQuery

q3 = StructSimilarityQuery(structure_search_type="file_url",
                           file_url="https://files.rcsb.org/view/4HHB.cif",
                           file_format="cif")
list(q3())

# If you want to upload your own structure file for similarity search, you can do so␣
↪by using the `file_path` parameter:
```

(continues on next page)

(continued from previous page)

```
q4 = StructSimilarityQuery(structure_search_type="file_upload",
                           file_path="/PATH/TO/FILE.cif",  # specify local model file␣
↪path
                           file_format="cif")
list(q4())
```

2.1.10 Structure Motif Query Examples

The PDB Archive can also be queried by using a "motif" found in these 3D structures. To perform this type of query, an entry_id or a file URL/path must be provided, along with residues (which are parts of 3D structures.) This is the bare minimum needed to make a search, but there are lots of other parameters that can be added to a Structure Motif Query (see full search schema).

To make a Structure Motif Query, you must first define anywhere from 2-10 "residues" that will be used in the query. Each individual residue has a Chain ID, Operator, Residue Number, and Exchanges (optional) that can be declared in that order using positonal arguments, or using the "chain_id", "struct_oper_id", and "label_seq_id" to define what parameter you are passing through. All 3 of the required parameters must be included, or the package will throw an AssertionError.

Each residue can only have a maximum of 4 Exchanges, and each query can only have 16 exchanges total. Violating any of these rules will cause the package to throw an AssertionError.

Examples of how to instantiate Residues can be found below. These can then be put into a list and passed through to a Structure Motif Query.

```
from rcsbsearchapi.search import StructureMotifResidue

# construct a Residue with a Chain ID of A, an operator of 1, a residue
# number of 192, and Exchanges of "LYS" and "HIS"
Res1 = StructureMotifResidue("A", "1", 192, ["LYS", "HIS"])
# as for what is a valid "Exchange", the package provides these as a literal,
# and they should be type checked.

# you can also specify the arguments:
# this query is the same as above.
Res2 = StructureMotifResidue(struct_oper_id="1", chain_id="A", exchanges=["LYS", "HIS
↪"], label_seq_id=192)

# after delcaring a minimum of 2 and as many as 10 residues, they can be passed into␣
↪a list for use in the query itself:
Res3 = StructureMotifResidue("A", "1", 162)  # exchanges are optional

ResList = [Res1, Res3]
```

From there, these Residues can be used in a query. As stated before, you can only include 2 - 10 residues in a query. If you fail to provide residues for a query, or provide the wrong amount, the package will throw a ValueError.

For a Structure Motif Query using an entry_id, the only other necessary value that must be passed into the query is the residue list. The default type of query is an entry_id query.

As this type of query has a lot of optional parameters, do *not* use positional arguments as more than likely an error will occur.

Below is an example of a basic entry_id Structure Motif Query, with the residues declared earlier:

```
from rcsbsearchapi.search import StructMotifQuery

q1 = StructMotifQuery(entry_id="2MNR", residue_ids=ResList)
list(q1())
```

Like with Structure Similarity Queries, a file url or filepath can also be provided to the program. These can take the place of an entry_id.

For a file url query, you *must* provide both a valid file URL (a string), and the file's file extension (also as a string). Failure to provide these elements correctly will cause the package to throw an AssertionError.

Below is an example of the same query as above, only this time providing a file url:

```
link = "https://files.rcsb.org/view/2MNR.cif"
q2 = StructMotifQuery(structure_search_type="file_url", url=link, file_extension="cif
↪", residue_ids=ResList)
# structure_search_type MUST be provided. A mismatched query type will cause an error.
↪
list(q2())
```

Like with Structure Similarity Queries, a filepath to a file may also be provided. This file must be a valid file accepted by the search API. A file extension must also be provided with the file upload.

The query would look something like this:

```
filepath = "/absolute/path/to/file.cif"
q3 = StructMotifQuery(structure_search_type="file_upload", file_path=filepath, file_
↪extension="cif", residue_ids=ResList)

list(q3())
```

There are many additional parameters that Structure Motif Query supports. These include a variety of features such as backbone distance tolerance, side chain distance tolerance, angle tolerance, RMSD cutoff, limits (stop searching after this many hits), atom pairing schemes, motif pruning strategy, allowed structures, and excluded structures. These can be mixed and matched as needed to make accurate and useful queries. All of these have some default value which is used when a parameter isn't provided. These parameters conform to the defaults used by the Search API.

Below will demonstrate how to define these parameters using non-positional arguments:

```
# specifying backbone distance tolerance: 0-3, default is 1
# allowed backbone distance tolerance in Angstrom.
backbone = StructMotifQuery(entry_id="2MNR", backbone_distance_tolerance=2, residue_
↪ids=ResList)
list(backbone())

# specifying sidechain distance tolerance: 0-3, default is 1
# allowed side-chain distance tolerance in Angstrom.
sidechain = StructMotifQuery(entry_id="2MNR", side_chain_distance_tolerance=2,␣
↪residue_ids=ResList)
list(sidechain())

# specifying angle tolerance: 0-3, default is 1
# allowed angle tolerance in multiples of 20 degrees.
angle = StructMotifQuery(entry_id="2MNR", angle_tolerance=2, residue_ids=ResList)
list(angle())

# specifying RMSD cutoff: >=0, default is 2
# Threshold above which hits will be filtered by RMSD
```

(continues on next page)

(continued from previous page)

```
rmsd = StructMotifQuery(entry_id="2MNR", rmsd_cutoff=1, residue_ids=ResList)
list(rmsd())

# specifying limit: >=0, default excluded
# Stop accepting results after this many hits.
limit = StructMotifQuery(entry_id="2MNR", limit=100, residue_ids=ResList)
list(limit())

# specifying atom pairing scheme, default = "SIDE_CHAIN"
# ENUM: "ALL", "BACKBONE", "SIDE_CHAIN", "PSUEDO_ATOMS"
# this is typechecked by a literal.
# Which atoms to consider to compute RMSD scores and transformations.
atom = StructMotifQuery(entry_id="2MNR", atom_pairing_scheme="ALL", residue_
↪ids=ResList)
list(atom())

# specifying motif pruning strategy, default = "KRUSKAL"
# ENUM: "NONE", "KRUSKAL"
# this is typechecked by a literal in the package.
# Specifies how many query motifs are "pruned". KRUSKAL leads to less stringent␣
↪queries, and faster results.
pruning = StructMotifQuery(entry_id="2MNR", motif_pruning_strategy="NONE", residue_
↪ids=ResList)
list(pruning())

# specifying allowed structures, default excluded
# specify the structures you wish to allow in the return result. As an example,
# we could only allow the results from the limited query we ran earlier.
allowed = StructMotifQuery(entry_id="2MNR", allowed_structures=list(limit()), residue_
↪ids=ResList)
list(allowed())

# specifying structures to exclude, default excluded
# specify structures to exclude from a query. We could, for example,
# exclude the results of the previous allowed query.
excluded = StructMotifQuery(entry_id="2MNR", excluded_structures=list(allowed()),␣
↪residue_ids=ResList)
list(excluded())
```

The Structure Motif Query can be used to make some very specific queries. Below is an example of a query that retrives occurances of the enolase superfamily, a group of proteins diverse in sequence and structure that are all capable of abstracting a proton from a carboxylic acid. Position-specific exchanges are crucial to represent this superfamily accurately.

```
Res1 = StructureMotifResidue("A", "1", 162, ["LYS", "HIS"])
Res2 = StructureMotifResidue("A", "1", 193)
Res3 = StructureMotifResidue("A", "1", 219)
Res4 = StructureMotifResidue("A", "1", 245, ["GLU", "ASP", "ASN"])
Res5 = StructureMotifResidue("A", "1", 295, ["HIS", "LYS"])

ResList = [Res1, Res2, Res3, Res4, Res5]

query = StructMotifQuery(entry_id="2MNR", residue_ids=ResList)

list(query())
```

2.1.11 Chemical Similarity Query

When you have unique chemical information (e.g., a chemical formula or descriptor) you can use this information to find chemical components (e.g., drugs, inhibitors, modified residues, or building blocks such as amino acids, nucleotides, or sugars), so that it is similar to the formula or descriptor used in the query (perhaps one or two atoms/groups are different), is part of a larger molecule (i.e., the specified formula/descriptor is a substructure), or is exactly or very closely matches the formula or descriptor used in the query.

The search can also be used to identify PDB structures that include the chemical component(s) which match or are similar to the query. These structures can then be examined to learn about the interactions of the component within the structure. More information on Chemical Similarity Queries can be found on the RCSB PDB Chemical Similarity Search page.

To do a Chemical Similarity query, you must first specify one of two possible query options which are formula and descriptors. Formula allows queries to be made by providing a chemical formula. Descriptors allow you to search by chemical notations for example. Each Query option has its own distinct set of parameters, but both options require a value.

The formula query option comes with a match subset parameter which allows users to search chemical components whose formula exactly match the query or matches any portion of the query. The descriptor query option comes with a descriptor type parameter and match type parameter. The descriptor type parameter specifies what type of descriptor the input value is. There are two options which are SMILES (Simplified Molecular Input Line Entry Specification) and InChI (International Chemical Identifier). The match type parameter has six options which are Similar Ligands (Quick Screen), Similar Ligands (Stereospecific), Similar Ligands (including Stereoisomers), Substructure (Stereospecific), Substructure (including Stereoisomers), and Exact match.

When doing Chemical Similarity Queries in this tool, it is important to note that by default the query option is set to formula and match subset is set to False. An example of how that looks like is below.

```
from rcsbsearchapi.search import ChemSimilarityQuery

# Basic query with default values: query type = formula and match subset = False
q1 = ChemSimilarityQuery(value="C12 H17 N4 O S")

# Same example but with all the parameters listed
q1 = ChemSimilarityQuery(value="C12 H17 N4 O S",
                         query_type="formula",
                         match_subset=False)
list(q1())
```

Below is are two examples of using query option descriptor. Both descriptor type parameters are also used.

```
from rcsbsearchapi.search import ChemSimilarityQuery

# Query with type = descriptor, descriptor type = SMILES, match type = similar␣
↪ligands (sterospecific) or graph-relaxed-stereo
q2 = ChemSimilarityQuery(value="Cc1c(sc[n+]1Cc2cnc(nc2N)C)CCO",
                         query_type="descriptor",
                         descriptor_type="SMILES",
                         match_type="graph-relaxed-stereo")
list(q2())
```

```
from rcsbsearchapi.search import ChemSimilarityQuery

# Query with type = descriptor, descriptor type = InChI, match type = substructure␣
↪(sterospecific) or sub-struct-graph-relaxed-stereo
q3 = ChemSimilarityQuery(value="InChI=1S/C13H10N2O4/c16-10-6-5-9(11(17)14-10)15-
↪12(18)7-3-1-2-4-8(7)13(15)19/h1-4,9H,5-6H2,(H,14,16,17)/t9-/m0/s1",
```

(continues on next page)

(continued from previous page)

```
                          query_type="descriptor",
                          descriptor_type="InChI",
                          match_type="sub-struct-graph-relaxed-stereo")
list(q3())
```

2.1.12 Faceted Query Examples

In order to group and perform calculations and statistics on PDB data by using a simple search query, you can use a faceted query (or facets). Facets arrange search results into categories (buckets) based on the requested field values. More information on Faceted Queries can be found here. All facets should be provided with `name`, `aggregation_type`, and `attribute` values. Depending on the aggregation type, other parameters must also be specified. The `facets()` function runs the query `q` using the specified facet(s), and returns a list of dictionaries:

```
from rcsbsearchapi.search import AttributeQuery, Facet, Range

q = AttributeQuery("rcsb_accession_info.initial_release_date", operator="greater",␣
↪value="2019-08-20")
q.facets(facets=Facet(name="Methods", aggregation_type="terms", attribute="exptl.
↪method"))
```

Term Facets

Terms faceting is a multi-bucket aggregation where buckets are dynamically built - one per unique value. We can specify the minimum count (`>= 0`) for a bucket to be returned using the parameter `min_interval_population` (default value `1`). We can also control the number of buckets returned (`<= 65336`) using the parameter `max_num_intervals` (default value `65336`).

```
# This is the default query, used by the RCSB Search API when no query is explicitly␣
↪specified.
# This default query will be used for most of the examples found below for faceted␣
↪queries.
base_q = AttributeQuery("rcsb_entry_info.structure_determination_methodology",␣
↪operator="exact_match", value="experimental")

base_q.facets(facets=Facet(name="Journals", aggregation_type="terms", attribute="rcsb_
↪primary_citation.rcsb_journal_abbrev", min_interval_population=1000))
```

Histogram Facets

Histogram facets build fixed-sized buckets (intervals) over numeric values. The size of the intervals must be specified in the parameter `interval`. We can also specify `min_interval_population` if desired.

```
base_q.facets(return_type="polymer_entity", facets=Facet(name="Formula Weight",␣
↪aggregation_type="histogram", attribute="rcsb_polymer_entity.formula_weight",␣
↪interval=50, min_interval_population=1))
```

Date Histogram Facets

Similar to histogram facets, date histogram facetes build buckets over date values. For date histogram aggregations, we must specify `interval="year"`. Again, we may also specify `min_interval_population`.

```
base_q.facets(facets=Facet(name="Release Date", aggregation_type="date_histogram",␣
↪attribute="rcsb_accession_info.initial_release_date", interval="year", min_interval_
↪population=1))
```

Range Facets

We can define the buckets ourselves by using range facets. In order to specify the ranges, we use the `Range` class. Note that the range includes the `start` value and excludes the `end` value (`include_lower` and `include_upper` should not be specified). If the `start` or `end` is omitted, the minimum or maximum boundaries will be used by default. The buckets should be provided as a list of `Range` objects to the `ranges` parameter.

```
base_q.facets(facets=Facet(name="Resolution Combined", aggregation_type="range",␣
↪attribute="rcsb_entry_info.resolution_combined", ranges=[Range(start=None,end=2),␣
↪Range(start=2, end=2.2), Range(start=2.2, end=2.4), Range(start=4.6, end=None)]))
```

Date Range Facets

Date range facets allow us to specify date values as bucket ranges, using date math expressions.

```
base_q.facets(facets=Facet(name="Release Date", aggregation_type="date_range",␣
↪attribute="rcsb_accession_info.initial_release_date", ranges=[Range(start=None,end=
↪"2020-06-01||-12M"), Range(start="2020-06-01", end="2020-06-01||+12M"), Range(start=
↪"2020-06-01||+12M", end=None)]))
```

Cardinality Facets

Cardinality facets return a single value: the count of distinct values returned for a given field. A `precision_threshold` (`<= 40000`, default value `40000`) may be specified.

```
base_q.facets(facets=Facet(name="Organism Names Count", aggregation_type="cardinality
↪", attribute="rcsb_entity_source_organism.ncbi_scientific_name"))
```

Multidimensional Facets

Complex, multi-dimensional aggregations are possible by specifying additional facets in the `nested_facets` parameter, as in the example below:

```
f1 = Facet(name="Polymer Entity Types", aggregation_type="terms", attribute="rcsb_
↪entry_info.selected_polymer_entity_types")
f2 = Facet(name="Release Date", aggregation_type="date_histogram", attribute="rcsb_
↪accession_info.initial_release_date", interval="year")
base_q.facets(facets=Facet(name="Experimental Method", aggregation_type="terms",␣
↪attribute="rcsb_entry_info.experimental_method", nested_facets=[f1, f2]))
```

Filter Facets

Filters allow us to filter documents that contribute to bucket count. Similar to queries, we can group several `TerminalFilters` into a single `GroupFilter`. We can combine a filter with a facet using the `FilterFacet` class. Terminal filters should specify an `attribute` and `operator`, as well as possible a `value` and whether or not it should be a `negation` and/or `case_sensitive`. Group filters should specify a `logical_operator` (which should be either `"and"` or `"or"`) and a list of filters (`nodes`) that should be combined. Finally, the `FilterFacet` should be provided with a filter and a (list of) facet(s). Here are some examples:

```
from rcsbsearchapi.search import TerminalFilter, GroupFilter, FilterFacet
tf1 = TerminalFilter(attribute="rcsb_polymer_instance_annotation.type", operator=
↪"exact_match", value="CATH")
tf2 = TerminalFilter(attribute="rcsb_polymer_instance_annotation.annotation_lineage.id
↪", operator="in", value=["2.140.10.30", "2.120.10.80"])
ff2 = FilterFacet(filters=tf2, facets=Facet("CATH Domains", "terms", "rcsb_polymer_
↪instance_annotation.annotation_lineage.id", min_interval_population=1))
ff1 = FilterFacet(filters=tf1, facets=ff2)
base_q.facets("polymer_instance", ff1)

tf1 = TerminalFilter(attribute="rcsb_struct_symmetry.kind", operator="exact_match",␣
↪value="Global Symmetry", negation=False)
f2 = Facet(name="ec_terms", aggregation_type="terms", attribute="rcsb_polymer_entity.
↪rcsb_ec_lineage.id")
f1 = Facet(name="sym_symbol_terms", aggregation_type="terms", attribute="rcsb_struct_
↪symmetry.symbol", nested_facets=f2)
ff = FilterFacet(filters=tf1, facets=f1)
q1 = AttributeQuery("rcsb_assembly_info.polymer_entity_count", operator="equals",␣
↪value=1)
q2 = AttributeQuery("rcsb_assembly_info.polymer_entity_instance_count", operator=
↪"greater", value=1)
q = q1 & q2
q.facets("assembly", ff)

tf1 = TerminalFilter(attribute="rcsb_polymer_entity_group_membership.aggregation_
↪method", operator="exact_match", value="sequence_identity")
tf2 = TerminalFilter(attribute="rcsb_polymer_entity_group_membership.similarity_cutoff
↪", operator="equals", value=100)
gf = GroupFilter(logical_operator="and", nodes=[tf1, tf2])
ff = FilterFacet(filters=gf, facets=Facet("Distinct Protein Sequence Count",
↪"cardinality", "rcsb_polymer_entity_group_membership.group_id"))
base_q.facets("polymer_entity", ff)
```

2.2 Sessions

The result of executing a query (either by calling it or using `exec()`) is a `Session` object. It implements `__iter__`, so it is usually treated just as an iterator of IDs.

Paging is handled transparently by the session, with additional API requests made lazily as needed. The page size can be controlled with the `rows` parameter.

```
first = next(iter(query(rows=1)))
```

2.2.1 Progress Bar

The `Session.iquery()` method provides a progress bar indicating the number of API requests being made. It requires the `tqdm` package be installed to track the progress of the query interactively.

```
results = query().iquery()
```

CHAPTER

THREE

API DOCUMENTATION

RCSB PDB Search API

class rcsbsearchapi.**Attr**(*attribute: str*, *type: Optional[str] = 'text'*)

A search attribute, e.g. "rcsb_entry_container_identifiers.entry_id"

Terminals can be constructed from Attr objects using either a functional syntax, which mirrors the API operators, or with python operators.

Rather than their normal bool return values, operators return Terminals.

Pre-instantiated attributes are available from the *rcsbsearchapi.rcsb_attributes* object. These are generally easier to use than constructing Attr objects by hand. A complete list of valid attributes is available in the schema.

- The *range* dictionary requires the following keys:
- "from" -> int
- "to" -> int
- "include_lower" -> bool
- "include_upper" -> bool

__contains__(*value: Union[str, List[str], rcsbsearchapi.search.Value[str], rcsbsearchapi.search.Value[List[str]]]*) → rcsbsearchapi.search.Terminal

Maps to contains_words or contains_phrase depending on the value passed.

- *"value" in attr* maps to *attr.contains_phrase("value")* for simple values.
- *["value"] in attr* maps to *attr.contains_words(["value"])* for lists and tuples.

__eq__(*value:* Attr) → bool

__eq__(*value: Union[str, int, float, datetime.date,* Value*[str],* Value*[int],* Value*[float],* Value*[date]]*) → rcsbsearchapi.search.Terminal

Return self==value.

__ge__(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.Terminal

Return self>=value.

__gt__(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.Terminal

Return self>value.

__le__(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.Terminal
Return self<=value.

__lt__(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.Terminal
Return self<value.

__ne__(*value:* Attr) → bool
__ne__(*value: Union[str, int, float, datetime.date,* Value*[str],* Value*[int],* Value*[float],* Value*[date]]*) → rcsbsearchapi.search.Terminal
Return self!=value.

__weakref__
list of weak references to the object (if defined)

contains_phrase(*value: Union[str, rcsbsearchapi.search.Value[str]]*) → rcsbsearchapi.search.AttributeQuery
Match an exact phrase

contains_words(*value: Union[str, rcsbsearchapi.search.Value[str], List[str], rcsbsearchapi.search.Value[List[str]]]*) → rcsbsearchapi.search.AttributeQuery
Match any word within the string.

Words are split at whitespace. All results which match any word are returned, with results matching more words sorted first.

equals(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.AttributeQuery
Attribute == *value*

exact_match(*value: Union[str, rcsbsearchapi.search.Value[str]]*) → rcsbsearchapi.search.AttributeQuery
Exact match with the value

exists() → rcsbsearchapi.search.AttributeQuery
Attribute is defined for the structure

greater(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.AttributeQuery
Attribute > *value*

greater_or_equal(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.AttributeQuery
Attribute >= *value*

in_(*value: Union[List[str], List[int], List[float], List[datetime.date], Tuple[str, ...], Tuple[int, ...], Tuple[float, ...], Tuple[datetime.date, ...], rcsbsearchapi.search.Value[List[str]], rcsbsearchapi.search.Value[List[int]], rcsbsearchapi.search.Value[List[float]], rcsbsearchapi.search.Value[List[datetime.date]], rcsbsearchapi.search.Value[Tuple[str, ...]], rcsbsearchapi.search.Value[Tuple[int, ...]], rcsbsearchapi.search.Value[Tuple[float, ...]], rcsbsearchapi.search.Value[Tuple[datetime.date, ...]]]*) → rcsbsearchapi.search.AttributeQuery
Attribute is contained in the list of values

less(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.AttributeQuery

Attribute < *value*

less_or_equal(*value: Union[int, float, datetime.date, rcsbsearchapi.search.Value[int], rcsbsearchapi.search.Value[float], rcsbsearchapi.search.Value[datetime.date]]*) → rcsbsearchapi.search.AttributeQuery
Attribute <= *value*

range(*value: Dict[str, Any]*) → rcsbsearchapi.search.AttributeQuery
Attribute is within the specified half-open range

Parameters **value** – lower and upper bounds *[a, b)*

class rcsbsearchapi.**Group**(*operator: typing_extensions.Literal[and, or], nodes: Iterable[rcsbsearchapi.search.Query] = ()*)
AND and OR combinations of queries

__and__(*other: rcsbsearchapi.search.Query*) → rcsbsearchapi.search.Query
Intersection: *a & b*

__invert__()
Negation: *~a*

__or__(*other: rcsbsearchapi.search.Query*) → rcsbsearchapi.search.Query
Union: *a | b*

_assign_ids(*node_id=0*) → Tuple[rcsbsearchapi.search.Query, int]
Assign node_ids sequentially for all terminal nodes

This is a helper for the *Query.assign_ids()* method

Parameters **node_id** – Id to assign to the first leaf of this query

Returns The modified query, with node_ids assigned node_id: The next available node_id

Return type query

to_dict()
Get dictionary representing this query

class rcsbsearchapi.**Query**
Base class for all types of queries.

Queries can be combined using set operators:

- *q1 & q2*: Intersection (AND)
- *q1 | q2*: Union (OR)
- *~q1*: Negation (NOT)
- *q1 - q2*: Difference (implemented as *q1 & ~q2*)
- *q1 ^ q2*: Symmetric difference (XOR, implemented as *(q1 & ~q2) | (~q1 & q2)*)

Note that only AND, OR, and negation of terminals are directly supported by the API, so other operations may be slower.

Queries can be executed by calling them as functions (*list(query())*) or using the exec function.

Queries are immutable, and all modifying functions return new instances.

__and__(*other: rcsbsearchapi.search.Query*) → rcsbsearchapi.search.Query
Intersection: *a & b*

__call__(*return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance, mol_definition] = 'entry'*, *rows: int = 10000*, *return_content_type: List[typing_extensions.Literal[experimental, computational]] = ['experimental']*, *results_verbosity: typing_extensions.Literal[compact, minimal, verbose] = 'compact'*) → rcsbsearchapi.search.Session
Evaluate this query and return an iterator of all result IDs

abstract __invert__() → rcsbsearchapi.search.Query
Negation: *~a*

__or__(*other: rcsbsearchapi.search.Query*) → rcsbsearchapi.search.Query
Union: *a | b*

__sub__(*other: rcsbsearchapi.search.Query*) → rcsbsearchapi.search.Query
Difference: *a - b*

__weakref__
list of weak references to the object (if defined)

__xor__(*other: rcsbsearchapi.search.Query*) → rcsbsearchapi.search.Query
Symmetric difference: *a ^ b*

abstract _assign_ids(*node_id=0*) → Tuple[rcsbsearchapi.search.Query, int]
Assign node_ids sequentially for all terminal nodes

This is a helper for the *Query.assign_ids()* method

Parameters **node_id** – Id to assign to the first leaf of this query

Returns The modified query, with node_ids assigned node_id: The next available node_id

Return type query

and_(*other:* Query) → *Query*
and_(*other: Union[str,* Attr*]*) → PartialQuery
Extend this query with an additional attribute via an AND

assign_ids() → rcsbsearchapi.search.Query
Assign node_ids sequentially for all terminal nodes

Returns the modified query, with node_ids assigned sequentially from 0

count(*return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance, mol_definition] = 'entry'*, *return_content_type: List[typing_extensions.Literal[experimental, computational]] = ['experimental']*) → int
Get the number of results found by this query

exec(*return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance, mol_definition] = 'entry'*, *rows: int = 10000*, *return_content_type: List[typing_extensions.Literal[experimental, computational]] = ['experimental']*, *results_verbosity: typing_extensions.Literal[compact, minimal, verbose] = 'compact'*) → rcsbsearchapi.search.Session
Evaluate this query and return an iterator of all result IDs

facets(*return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance, mol_definition] = 'entry'*, *facets: Optional[Union[rcsbsearchapi.search.Facet, rcsbsearchapi.search.FilterFacet, List[Union[rcsbsearchapi.search.Facet, rcsbsearchapi.search.FilterFacet]]]] = None*) → List
Perform a facets query and return the buckets

or_(*other:* Query) → *Query*

or_(*other: Union[str, Attr]*) → PartialQuery
Extend this query with an additional attribute via an OR

abstract to_dict() → Dict
Get dictionary representing this query

to_json() → str
Get JSON string of this query

class rcsbsearchapi.**Session**(*query: rcsbsearchapi.search.Query*, *return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance, mol_definition] = 'entry'*, *rows: int = 10000*, *return_content_type: List[typing_extensions.Literal[experimental, computational]] = ['experimental']*, *results_verbosity: typing_extensions.Literal[compact, minimal, verbose] = 'compact'*, *facets: Optional[Union[rcsbsearchapi.search.Facet, rcsbsearchapi.search.FilterFacet, List[Union[rcsbsearchapi.search.Facet, rcsbsearchapi.search.FilterFacet]]]] = None*)
A single query session.

Handles paging the query and parsing results

__init__(*query: rcsbsearchapi.search.Query*, *return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance, mol_definition] = 'entry'*, *rows: int = 10000*, *return_content_type: List[typing_extensions.Literal[experimental, computational]] = ['experimental']*, *results_verbosity: typing_extensions.Literal[compact, minimal, verbose] = 'compact'*, *facets: Optional[Union[rcsbsearchapi.search.Facet, rcsbsearchapi.search.FilterFacet, List[Union[rcsbsearchapi.search.Facet, rcsbsearchapi.search.FilterFacet]]]] = None*)
Initialize self. See help(type(self)) for accurate signature.

__iter__() → Iterator[str]
Generator for all results as a list of identifiers

__weakref__
list of weak references to the object (if defined)

static _extract_identifiers(*query_json: Optional[Dict]*) → List[str]
Extract identifiers from a JSON response

_make_params(*start=0*)
Generate GET parameters as a dict

_single_query(*start=0*) → Optional[Dict]
Fires a single query

iquery(*limit: Optional[int] = None*) → List[str]
Evaluate the query and display an interactive progress bar.

Requires tqdm.

static make_uuid() → str
Create a new UUID to identify a query

rcsb_query_builder_url() → str
URL to view this query on the RCSB PDB website query builder

rcsb_query_editor_url() → str
URL to edit this query in the RCSB PDB query editor

class rcsbsearchapi.**Terminal**(*service: str*, *params: Dict[str, Any]*, *node_id: int = 0*)
A terminal query node.

Used for doing various types of searches. Accepts a service type and a dictionary of parameters. The set of parameters differs for different search services.

Terminal can be built by passing in a service and parameter dictionary, but it's tedious work. Typically, it's built by child classes that each represent a unique type of search. This allows for more concise searching.

Examples

```
>>> Terminal("full_text", {"value": "protease"})
>>> Terminal("text", {"attribute": "rcsb_id", "operator": "in", "negation": False,
↪ "value": ["5T89, "1TIM"]})
```

__invert__()
Negation: *~a*

_assign_ids(*node_id=0*) → Tuple[rcsbsearchapi.search.Query, int]
Assign node_ids sequentially for all terminal nodes

This is a helper for the *Query.assign_ids()* method

Parameters **node_id** – Id to assign to the first leaf of this query

Returns The modified query, with node_ids assigned node_id: The next available node_id

Return type query

to_dict()
Get dictionary representing this query

class rcsbsearchapi.**TextQuery**(*value: str*)
Special case of a Terminal for free-text queries

__init__(*value: str*)
Search for the string value anywhere in the text

Parameters **value** – free-text query

class rcsbsearchapi.**Value**(*value: T*)
Represents a value in a query.

In most cases values are unnecessary and can be replaced directly by the python value.

Values can also be used if the Attr object appears on the right:

Value("4HHB") == Attr("rcsb_entry_container_identifiers.entry_id")

__eq__(*attr: Value*) → bool
__eq__(*attr: rcsbsearchapi.search.Attr*) → rcsbsearchapi.search.Terminal
Return self==value.

__ge__(*attr: rcsbsearchapi.search.Attr*) → rcsbsearchapi.search.Terminal
Return self>=value.

__gt__(*attr: rcsbsearchapi.search.Attr*) → rcsbsearchapi.search.Terminal
Return self>value.

__le__(*attr: rcsbsearchapi.search.Attr*) → rcsbsearchapi.search.Terminal
Return self<=value.

__lt__ (*attr: rcsbsearchapi.search.Attr*) → rcsbsearchapi.search.Terminal
Return self<value.

__ne__ (*attr:* Value) → bool
__ne__ (*attr: rcsbsearchapi.search.Attr*) → rcsbsearchapi.search.Terminal
Return self!=value.

__weakref__
list of weak references to the object (if defined)

rcsbsearchapi.**rcsb_attributes: SchemaGroup = <rcsbsearchapi.schema.SchemaGroup object>**
Object with all known RCSB PDB attributes.

This is provided to ease autocompletion as compared to creating Attr objects from strings. For example,

```
rcsb_attributes.rcsb_nonpolymer_instance_feature_summary.chem_id
```

is equivalent to

```
Attr('rcsb_nonpolymer_instance_feature_summary.chem_id')
```

All attributes in *rcsb_attributes* can be iterated over.

```
>>> [a for a in rcsb_attributes if "stoichiometry" in a.attribute]
[Attr(attribute='rcsb_struct_symmetry.stoichiometry')]
```

Attributes matching a regular expression can also be filtered:

```
>>> list(rcsb_attributes.search('rcsb.*stoichiometry'))
[Attr(attribute='rcsb_struct_symmetry.stoichiometry')]a
```

CHAPTER

FOUR

AVAILABILITY

Get it from PyPI:

```
pip install rcsbsearchapi
```

Or, download from GitHub

CHAPTER

FIVE

LICENSE

Code is licensed under the BSD 3-clause license. See the LICENSE for details.

CHAPTER

SIX

CITING

Please cite the rcsbsearchapi package by URL:

> https://rcsbsearchapi.readthedocs.io

You should also cite the RCSB PDB service this package utilizes:

> Yana Rose, Jose M. Duarte, Robert Lowe, Joan Segura, Chunxiao Bi, Charmi Bhikadiya, Li Chen, Alexander S. Rose, Sebastian Bittrich, Stephen K. Burley, John D. Westbrook. RCSB Protein Data Bank: Architectural Advances Towards Integrated Searching and Efficient Access to Macromolecular Structure Data from the PDB Archive, Journal of Molecular Biology, 2020. DOI: 10.1016/j.jmb.2020.11.003

PYTHON MODULE INDEX

r

INDEX

Symbols

A

C

E

F

G

I

L

M

O

Q

R

S

T

V