rcsbsearch

Release 0.3.0-dev0

Spencer Bliven

May 05, 2021

CONTENTS

The `rcsbsearch` package provides a python interface to the RCSB 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 rcsbsearch
```

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 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 rcsbsearch import TextQuery
from rcsbsearch 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  # AND of all queries

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

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

1.2.2 Fluent Example

Here is the same example using the fluent syntax

```
from rcsbsearch import Attr, TextQuery

# Start with a Attr or TextQuery, then add terms
results = TextQuery('"heat-shock transcription factor"') \
    .and_("rcsb_struct_symmetry.symbol").exact_match("C2") \
    .and_("rcsb_struct_symmetry.kind").exact_match("Global Symmetry") \
    .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)
```

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 rcsbsearch import TextQuery
from rcsbsearch 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 schema.

`Terminal`s are combined into `Group`s 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"))
```

2.2 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 rcsbsearch import TextQuery

# Start with a Attr or TextQuery, then add terms
results = TextQuery('"heat-shock transcription factor"') \
    .and_("rcsb_struct_symmetry.symbol").exact_match("C2") \
    .and_("rcsb_struct_symmetry.kind").exact_match("Global Symmetry") \
    .and_("rcsb_entry_info.polymer_entity_count_DNA").greater_or_equal(1) \
    .exec("assembly")
```

2.3 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)))
```

2.3.1 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 Search API

class rcsbsearch.**Attr**(*attribute: str*)

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 *rcsbsearch.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.

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

Return self==value.

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

Return self>=value.

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

Return self>value.

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

Return self<=value.

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

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

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

contains_words(*value: Union[str, rcsbsearch.search.Value[str], List[str], rcsbsearch.search.Value[List[str]]]*) → rcsbsearch.search.Terminal
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, rcsbsearch.search.Value[int], rcsbsearch.search.Value[float], rcsbsearch.search.Value[datetime.date]]*) → rcsbsearch.search.Terminal
Attribute == *value*

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

exists() → rcsbsearch.search.Terminal
Attribute is defined for the structure

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

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

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

less(*value: Union[int, float, datetime.date, rcsbsearch.search.Value[int], rcsbsearch.search.Value[float], rcsbsearch.search.Value[datetime.date]]*) → rcsbsearch.search.Terminal
Attribute < *value*

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

range(*value: Union[List[int], Tuple[int, int]]*) → rcsbsearch.search.Terminal
Attribute is within the specified half-open range

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

range_closed(*value: Union[List[int], Tuple[int, int], rcsbsearch.search.Value[List[int]], rcsbsearch.search.Value[Tuple[int, int]]]*) → rcsbsearch.search.Terminal
Attribute is within the specified closed range

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

class rcsbsearch.**Group**(*operator: typing_extensions.Literal[and, or], nodes: Iterable[rcsbsearch.search.Query] = ()*)

AND and OR combinations of queries

__and__(*other: rcsbsearch.search.Query*) → rcsbsearch.search.Query

Intersection: *a & b*

__invert__()

Negation: *~a*

__or__(*other: rcsbsearch.search.Query*) → rcsbsearch.search.Query

Union: *a | b*

_assign_ids(*node_id=0*) → Tuple[rcsbsearch.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 rcsbsearch.**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: rcsbsearch.search.Query*) → rcsbsearch.search.Query

Intersection: *a & b*

__call__(*return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance] = 'entry', rows: int = 100*) → rcsbsearch.search.Session

Evaluate this query and return an iterator of all result IDs

abstract __invert__() → rcsbsearch.search.Query

Negation: *~a*

__or__(*other: rcsbsearch.search.Query*) → rcsbsearch.search.Query

Union: *a | b*

__sub__(*other: rcsbsearch.search.Query*) → rcsbsearch.search.Query

Difference: *a - b*

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

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

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

Returns the modified query, with node_ids assigned sequentially from 0

exec(*return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance] = 'entry'*, *rows: int = 100*) → rcsbsearch.search.Session
Evaluate this query and return an iterator of all result IDs

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 rcsbsearch.**Session**(*query: rcsbsearch.search.Query*, *return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance] = 'entry'*, *rows: int = 100*)
A single query session.

Handles paging the query and parsing results

__init__(*query: rcsbsearch.search.Query*, *return_type: typing_extensions.Literal[entry, assembly, polymer_entity, non_polymer_entity, polymer_instance] = 'entry'*, *rows: int = 100*)
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 website query builder

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

class rcsbsearch.**Terminal**(*attribute: Optional[str] = None*, *operator: Optional[str] = None*, *value: Optional[Union[str, int, float, datetime.date, List[str], List[int], List[float], List[datetime.date], Tuple[str, ...], Tuple[int, ...], Tuple[float, ...], Tuple[datetime.date, ...]]] = None*, *service: str = 'text'*, *negation: bool = False*, *node_id: int = 0*)
A terminal query node.

Terminals are simple predicates comparing some *attribute* of a structure to a value.

Examples

```
>>> Terminal("exptl.method", "exact_match", "X-RAY DIFFRACTION")
>>> Terminal("rcsb_id", "in", ["5T89", "1TIM"])
>>> Terminal(value="tubulin")
```

A full list of attributes is available in the schema. Operators are documented here.

The *Attr* class provides a more pythonic way of constructing Terminals.

__invert__()
Negation: *~a*

__str__()
Return a simplified string representation

Examples

```
>>> Terminal("attr", "op", "val")
>>> ~Terminal(value="val")
```

_assign_ids(*node_id=0*) → Tuple[rcsbsearch.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 rcsbsearch.**TextQuery**(*value: str*, *negation: bool = False*)
Special case of a Terminal for free-text queries

__init__(*value: str*, *negation: bool = False*)
Search for the string value anywhere in the text

Parameters

- **value** – free-text query
- **negation** – find structures without the pattern

class rcsbsearch.**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: rcsbsearch.search.Attr*) → rcsbsearch.search.Terminal
Return self==value.

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

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

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

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

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

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

rcsbsearch.**rcsb_attributes: SchemaGroup = <rcsbsearch.schema.SchemaGroup object>**
Object with all known RCSB 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 rcsbsearch
```

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 rcsbsearch package by URL:

> https://rcsbsearch.readthedocs.io

You should also cite the RCSB 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

G

I

L

M

O

Q

R

S

T

V