lXtractor

Release 0.1.5

iReveguk

Mar 21, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

CHAPTER

TWO

EXAMPLES

CHAPTER

THREE

API REFERENCE

3.1 lXtractor package

3.1.1 lXtractor.core package

lXtractor.core.alignment module

A module handling multiple sequence alignments.

class lXtractor.core.alignment.**Alignment**(*seqs*, *add_method=<function mafft_add>*, *align_method=<function mafft_align>*)

Bases: `object`

An MSA resource: a collection of aligned sequences.

__init__(*seqs*, *add_method=<function mafft_add>*, *align_method=<function mafft_align>*)

Parameters

- **seqs** (*Iterable[tuple[str, str]]*) – An iterable with (id, _seq) pairs.
- **add_method** (AddMethod) – A callable adding sequences. Check the type for a signature.
- **align_method** (AlignMethod) – A callable aligning sequences.

add(*other*)

Add sequences to existing ones using *add()*. This is similar to *align()* but automatically adds the aligned seqs.

```
>>> a = Alignment([('A', 'ABCD'), ('X', 'XXXX')])
>>> aa = a.add(('Y', 'ABXD'))
>>> aa.shape
(3, 4)
```

Parameters

other (*abc.Iterable[_ST] | _ST | Alignment*) – A sequence, iterable over sequences, or another *Alignment*.

Returns

A new *Alignment* object with added sequences.

Return type

t.Self

align(*seq*)

Align (add) sequences to this alignment via *add_method*.

```
>>> a = Alignment([('A', 'ABCD'), ('X', 'XXXX')])
>>> aa = a.align(('Y', 'ABXD'))
>>> aa.shape
(1, 4)
>>> aa.seqs
[('Y', 'ABXD')]
```

Parameters

seq (*abc.Iterable[_ST] | _ST | Alignment*) – A sequence, iterable over sequences, or another *Alignment*.

Returns

A new alignment object with sequences from *_seq*. The original number of columns should be preserved, which is true when using the default *add_method*.

Return type

t.Self

annotate(*objs*, *map_name*, *accept_fn=None*, ***kwargs*)

This function "annotates" sequence segments using MSA.

Namely, it adds each sequence of the provided chain-type objects to sequences currently present in this MSA via *add_method*. The latter is expected to preserve the original number of MSA columns, whereas potentially cutting the original sequence, thereby defining MSA-imposed boundaries. These are used to extract a child object using `spawn_child` method, which will have the corresponding MSA numbering written under *map_name*.

Parameters

- **objs** (*abc.Iterable[_CT]*) – An iterable over chain-type objects.
- **map_name** (*str*) – A name to use for storing the derived MSA numbering map.
- **accept_fn** (*abc.Callable[[_CT], bool] | None*) – A function accepting a chain-type object and returning a boolean value indicating whether the spawn child sequence should be preserved.
- **kwargs** – Additional keyword arguments passed to the `spawn_child()` method.

Returns

An iterator over spawned child objects. These are automatically stored under the `children` attribute of each chain-type object, in which case it's safe to simply consume the returned iterator.

filter(*fn*)

Filter alignment sequences.

Parameters

fn (*SeqFilter*) – A function accepting a sequence – (name, _seq) pair – and returning a boolean.

Returns

A new *Alignment* object with filtered sequences.

Return type

t.Self

filter_gaps(*max_frac=1.0*, *dim=0*)

Filter sequences or alignment columns having >= *max_frac* of gaps.

```
>>> a = Alignment([('A', 'AB---'), ('X', 'XXXX-'), ('Y', 'YYYY-')])
```

By default, the *max_frac* gaps is 1.0, which would remove solely gap-only sequences.

```
>>> aa = a.filter_gaps(dim=0)
>>> aa == a
True
```

Specifying *max_frac* removes sequences with over 50% gaps.

```
>>> aa = a.filter_gaps(dim=0, max_frac=0.5)
>>> 'A' not in aa
True
```

The last column is removed.

```
>>> a.filter_gaps(dim=1).shape
(3, 4)
```

Parameters

- **max_frac** (*float*) – a maximum fraction of allowed gaps in a sequence or a column.
- **dim** (*int*) – 0 for sequences, 1 for columns.

Returns

A new *Alignment* object with filtered sequences or columns.

Return type

t.Self

itercols(*, *join=True*)

Iterate over the Alignment columns.

```
>>> a = Alignment([('A', 'ABCD'), ('X', 'XXXX')])
>>> list(a.itercols())
['AX', 'BX', 'CX', 'DX']
```

Parameters

join (*bool*) – Join columns into a string.

Returns

An iterator over columns.

Return type

Iterator[str] | *Iterator*[list[str]]

classmethod make(*seqs*, *method=<function mafft_align>*, *add_method=<function mafft_add>*, *align_method=<function mafft_align>*)

Create a new alignment from a collection of unaligned sequences. For aligned sequences, please utilize *read()*.

Parameters

- **seqs** (*Iterable[tuple[str, str]]*) – An iterable over (header, _seq) objects.
- **method** (AlignMethod) – A callable accepting unaligned sequences and returning the aligned ones.
- **add_method** (AddMethod) – A sequence addition method for a new *Alignment* object.
- **align_method** (AlignMethod) – An alignment method for a new *Alignment* object.

Returns
An alignment created from aligned *seqs*.

Return type
Alignment

map(*fn*)

Map a function to sequences.

```
>>> a = Alignment([('A', 'AB---')])
>>> a.map(lambda x: (x[0].lower(), x[1].replace('-', '*'))).seqs
[('a', 'AB***')]
```

Parameters
fn (SeqMapper) – A callable accepting and returning a sequence.

Returns
A new *Alignment* object.

Return type
t.Self

classmethod read(*inp*, *read_method=<function read_fasta>*, *add_method=<function mafft_add>*, *align_method=<function mafft_align>*)

Read sequences and create an alignment.

Parameters

- **inp** (*Path | TextIOBase | abc.Iterable[str]*) – A Path to aligned sequences, or a file handle, or iterable over file lines.
- **read_method** (SeqReader) – A method accepting *inp* and returning an iterable over pairs (header, _seq). By default, it's `read_fasta()`. Hence, the default expected format is fasta.
- **add_method** (AddMethod) – A sequence addition method for a new *Alignment* object.
- **align_method** (AlignMethod) – An alignment method for a new *Alignment* object.

Returns
An alignment with sequences read parsed from the provided input.

Return type
t.Self

classmethod read_make(*inp*, *read_method=<function read_fasta>*, *add_method=<function mafft_add>*, *align_method=<function mafft_align>*)

A shortcut combining *read()* and *make()*.

It parses sequences from *inp*, aligns them and creates
the *Alignment* object.

Parameters

- **inp** (*Path | TextIOBase | abc.Iterable[str]*) – A Path to aligned sequences, or a file handle, or iterable over file lines.
- **read_method** (SeqReader) – A method accepting *inp* and returning an iterable over pairs (header, _seq). By default, it's `read_fasta()`. Hence, the default expected format is fasta.
- **add_method** (AddMethod) – A sequence addition method for a new *Alignment* object.
- **align_method** (AlignMethod) – An alignment method for a new *Alignment* object.

Returns
An alignment from parsed and aligned *inp* sequences.

Return type
t.Self

realign()

Realign sequences in *seqs* using *align_method*.

Returns
A new *Alignment* object with realigned sequences.

remove(*item*, *error_if_missing=True*, *realign=False*)

Remove a sequence or collection of sequences.

```
>>> a = Alignment([('A', 'ABCD-'), ('X', 'XXXX-'), ('Y', 'YYYYY')])
>>> aa = a.remove('A')
>>> 'A' in aa
False
>>> aa = a.remove(('Y', 'YYYYY'))
>>> aa.shape
(2, 5)
>>> aa = a.remove(('Y', 'YYYYY'), realign=True)
>>> aa.shape
(2, 4)
>>> aa['A']
'ABCD'
>>> aa = a.remove(['X', 'Y'])
>>> aa.shape
(1, 5)
```

Parameters

- **item** (*str | _ST | t.Iterable[str] | t.Iterable[_ST]*) – One of the following:
  - A `str`: a sequence's name.
  - A pair `(str, str)` – a name with the sequence itself.
  - An iterable over sequence enames or pairs (not mixed!)
- **error_if_missing** (*bool*) – Raise an error if any of the items are missing.
- **realign** (*bool*) – Realign seqs after removal.

Returns
A new *Alignment* object with the remaining sequences.

Return type
t.Self

slice(*start*, *stop*, *step=None*)

Slice alignment columns.

```
>>> a = Alignment([('A', 'ABCD'), ('X', 'XXXX')])
>>> aa = a.slice(1, 2)
>>> aa.shape == (2, 2)
True
>>>
>>> aa.seqs[0]
('A', 'AB')
>>> aa = a.slice(-4, 10)
>>> aa.seqs[0]
('A', 'ABCD')
```

To add the aligned sequences to the existing ones, use + or *add()*:

```
>>> aaa = a + aa
>>> aaa.shape
(3, 4)
```

Parameters

- **start** (*int*) – Start coordinate, boundaries inclusive.
- **stop** (*int*) – Stop coordinate, boundaries inclusive.
- **step** (*int | None*) – Step for slicing, i.e., take every column separated by *step - 1* number of columns.

Returns

A new alignment with sequences subset according to the slicing params.

Return type

t.Self

write(*out*, *write_method=<function write_fasta>*)

Write an alignment.

Parameters

- **out** (*Path | SupportsWrite*) – Any object with the *write* method.
- **write_method** (*SeqWriter*) – The writing function itself, accepting sequences and *out*. By default, use *read_fasta* to write in fasta format.

Returns

Nothing.

Return type

None

add_method: *AddMethod*

align_method: *AlignMethod*

seqs: list[tuple[str, str]]

property shape: tuple[int, int]

Returns
(# sequences, # columns)

lXtractor.core.base module

Base classes, commong types and functions for the *core* module.

class lXtractor.core.base.**AbstractResource**(*resource_path*, *resource_name*)

Bases: object

Abstract base class defining basic interface any resource must provide.

__init__(*resource_path*, *resource_name*)

Parameters

- **resource_path** (*str | Path*) – Path to parsed resource data.
- **resource_name** (*str | None*) – Resource's name.

abstract **dump**(*path*)

Save the resource under the given *path*.

abstract **fetch**(*url*)

Download the resource.

abstract **parse**()

Parse the read resource, so it's ready for usage.

abstract **read**()

Read the resource using the resource_path

class lXtractor.core.base.**AddMethod**(**args*, ***kwargs*)

Bases: Protocol

A callable to add sequences to the aligned ones, preserving the alignment length.

__call__(*msa*, *seqs*)

Call self as a function.

Return type
Iterable[tuple[str, str]]

__init__(**args*, ***kwargs*)

class lXtractor.core.base.**AlignMethod**(**args*, ***kwargs*)

Bases: Protocol

A callable to align arbitrary sequences.

__call__(*seqs*)

Call self as a function.

Return type
Iterable[tuple[str, str]]

__init__(**args*, ***kwargs*)

class lXtractor.core.base.**ApplyT**(**args*, ***kwargs*)

Bases: Protocol[T]

__call__(*x*)

Call self as a function.

Return type
T

__init__(**args*, ***kwargs*)

class lXtractor.core.base.**ApplyTWithArgs**(**args*, ***kwargs*)

Bases: Protocol[T]

__call__(*x*, **args*, ***kwargs*)

Call self as a function.

Return type
T

__init__(**args*, ***kwargs*)

class lXtractor.core.base.**FilterT**(**args*, ***kwargs*)

Bases: Protocol[T]

__call__(*x*)

Call self as a function.

Return type
bool

__init__(**args*, ***kwargs*)

class lXtractor.core.base.**NamedTupleT**(**args*, ***kwargs*)

Bases: Protocol, Iterable

__init__(**args*, ***kwargs*)

class lXtractor.core.base.**Ord**(**args*, ***kwargs*)

Bases: Protocol[_T]

Any objects defining comparison operators.

__init__(**args*, ***kwargs*)

class lXtractor.core.base.**ResNameDict**

Bases: UserDict

A dictionary providing mapping between PDB residue names and their one-letter codes. The mapping was parsed from the CCD and can be obtained by calling `lXtractor.ext.ccd.CCD.make_res_name_map()`.

```
>>> d = ResNameDict()
>>> assert d['ALA'] == 'A'
```

__init__()

class lXtractor.core.base.SeqFilter(*args, **kwargs)

Bases: Protocol

A callable accepting a pair (header, _seq) and returning a boolean.

__call__(seq, **kwargs)

Call self as a function.

Return type
bool

__init__(*args, **kwargs)

class lXtractor.core.base.SeqMapper(*args, **kwargs)

Bases: Protocol

A callable accepting and returning a pair (header, _seq).

__call__(seq, **kwargs)

Call self as a function.

Return type
tuple[str, str]

__init__(*args, **kwargs)

class lXtractor.core.base.SeqReader(*args, **kwargs)

Bases: Protocol

A callable reading sequences into tuples of (header, _seq) pairs.

__call__(inp)

Call self as a function.

Return type
Iterable[tuple[str, str]]

__init__(*args, **kwargs)

class lXtractor.core.base.SeqWriter(*args, **kwargs)

Bases: Protocol

A callable writing (header, _seq) pairs to disk.

__call__(inp, out)

Call self as a function.

__init__(*args, **kwargs)

class lXtractor.core.base.SupportsWrite(*args, **kwargs)

Bases: Protocol

Any object with the *write* method.

__init__(*args, **kwargs)

write(data)

Write the supplied data.

class lXtractor.core.base.**UrlGetter**(**args*, ***kwargs*)

Bases: Protocol

A callable accepting some string arguments and turning them into a valid url.

__call__(**args*)

Call self as a function.

Return type
str

__init__(**args*, ***kwargs*)

lXtractor.core.config module

A module encompassing various settings of lXtractor objects.

class lXtractor.core.config.**AtomMark**(*value*)

Bases: IntFlag

The atom categories. Some categories may be combined, e.g., LIGAND | PEP is another valid category denoting ligand peptide atoms.

CARB: int = 32

Carbohydrate polymer atoms.

COVALENT: int = 64

Covalent polymer modifications including ligands.

LIGAND: int = 4

Ligand atom. If not combined with PEP, NUC, or CARB, this category denotes non-polymer (small molecule) single-residue ligands.

NUC: int = 16

Nucleotide polymer atoms.

PEP: int = 8

Peptide polymer atoms.

SOLVENT: int = 2

Solvent atom.

UNK: int = 1

Unknown atom.

class lXtractor.core.config.**Config**(*default_config_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtra*
user_config_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtracto

Bases: UserDict

A configuration management class.

This class facilitates the loading and saving of configuration settings, with a user-specified configuration overriding the default settings.

Parameters

- **default_config_path** (*str | Path*) – The path to the default config file. This is a reference default settings, which can be used to reset user settings if needed.

- **user_config_path** (*str | Path*) – The path to the user configuration file. This file is stored internally and can be modified by a user to provide permanent settings.

Loading and mofifying the config:

```
>>> cfg = Config()
>>> list(cfg.keys())[:2]
['bonds', 'colnames']
>>> cfg['bonds']['non_covalent_upper']
5.0
>>> cfg['bonds']['non_covalent_upper'] = 6
```

Equivalently, one can update the config by a local JSON file or dict:

```
>>> cfg.update_with({'bonds': {'non_covalent_upper': 4}})
>>> assert cfg['bonds']['non_covalent_upper'] == 4
```

The changes can be stored internally and loaded automatically in the future:

```
>>> cfg.save()
>>> cfg = Config()
>>> assert cfg['bonds']['non_covalent_upper'] == 4
```

To restore default settings:

```
>>> cfg.reset_to_defaults()
>>> cfg.clear_user_config()
```

__init__(*default_config_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtractor/checkouts/latest/lXtract*
user_config_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtractor/checkouts/latest/lXtractor/

clear_user_config()

Clear the contents of the locally stored user config file.

reload()

Load the configuration from files.

reset_to_defaults()

Reset the configuration to the default settings.

save(*user_config_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtractor/checkouts/latest/lXtractor/resou*

Save the current configuration. By default, will store the configuration internally. This stored configuration will be loaded automatically on top of the default configuration.

Parameters

user_config_path (*str | Path*) – The path where to save the user configuration file.

Raises

ValueError – If the user config path is not provided.

temporary_namespace()

A context manager for a temporary config namespace.

Within this context, changes to the config are allowed, but will be reverted back to the original config once the context is exited.

Example:

```
>>> cfg = Config()
>>> with cfg.temporary_namespace():
...     cfg['bonds']['non_covalent_upper'] = 10
...     # Do some stuff with the temporary config...
... # Config is reverted back to original state here
>>> assert cfg['bonds']['non_covalent_upper'] != 10
```

update_with(*other*)

lXtractor.core.config.**serialize_json_value**(*obj*)

Recursively convert objects to a JSON-serializable form.

lXtractor.core.exceptions module

exception lXtractor.core.exceptions.**AmbiguousData**

Bases: ValueError

exception lXtractor.core.exceptions.**AmbiguousMapping**

Bases: ValueError

exception lXtractor.core.exceptions.**ConfigError**

Bases: ValueError

Some configuration problem.

exception lXtractor.core.exceptions.**FailedCalculation**

Bases: RuntimeError

exception lXtractor.core.exceptions.**FormatError**

Bases: ValueError

exception lXtractor.core.exceptions.**InitError**

Bases: ValueError

A broad category exception for problems with an object's initialization

exception lXtractor.core.exceptions.**LengthMismatch**

Bases: ValueError

exception lXtractor.core.exceptions.**MissingData**

Bases: ValueError

exception lXtractor.core.exceptions.**NoOverlap**

Bases: ValueError

exception lXtractor.core.exceptions.**OverlapError**

Bases: ValueError

exception lXtractor.core.exceptions.**ParsingError**

Bases: ValueError

IXtractor.core.ligand module

class lXtractor.core.ligand.**Ligand**(*parent*, *mask*, *contact_mask*, *ligand_idx*, *dist*, *meta=None*)

Bases: `object`

Ligand object is a part of the structure falling under certain criteria.

Namely, a ligand is a non-polymer and non-solvent molecule or a single monomer. Such ligands will be designated using the format:

```
{res_name}_{res_id}:{chain_id}<-({parent})
```

If a ligand contains multiple monomers, by convention, this is a polymer ligand. Such ligands should be named using the first letter of the polymer type; one of the `("p", "n", "c")`. In this case, it's ID will be of the following format:

```
{polymer_type}_{min_res_id}-{max_res_id}:{chain_id}<-({parent})
```

This information is provided by *meta* and shouldn't be changed. However, any additional fields can be stored in *meta* which will be retrieved when constructing *summary()*.

Attributes *mask* and *contact_mask* are boolean masks allowing to obtain ligand and ligand-contacting atoms from *parent*.

..seealso ::
make_ligand() to initialize a new ligand in an easy way.

__init__(*parent*, *mask*, *contact_mask*, *ligand_idx*, *dist*, *meta=None*)

is_locally_connected(*mask*)

Check whether this ligand is connected to a subset of parent atoms.

Parameters
mask (*ndarray*) – A boolean mask to filter parent atoms.

Returns
`True` if the ligand has at least *min_atom_connections* to *parent* substructure imposed by the provided *mask*.

Return type
bool

summary(*meta=True*)

Return type
Series

property array: AtomArray

Returns
An array of ligand atoms within *parent*.

property chain_id: str

Returns
Ligand chain ID.

contact_mask: np.ndarray

A boolean mask such that when applied to the parent, subsets the latter to its ligand-contacting atoms.

dist

An array of distances for each ligand-contacting parent's atom.

property id: str

is_polymer

ligand_idx

An integer array with indices pointing to ligand atoms contacting the parent structure.

mask

A boolean mask such that when applied to the parent, subsets the latter to the ligand residues.

meta

A dictionary of meta info.

parent: *GenericStructure*

Parent structure.

property parent_contact_atoms: AtomArray

Returns

An array of ligand-contacting atoms within *parent*.

property parent_contact_chains: set[str]

Returns

A set of chain IDs involved in forming contacts with ligand.

property res_id: str

Returns

Ligand residue number.

property res_name: str

Returns

Ligand residue name.

lXtractor.core.ligand.**ligands_from_atom_marks**(*structure*)

Return type

abc.Generator[*Ligand*, None, None]

lXtractor.core.ligand.**make_ligand**(*m_lig*, *m_pol*, *structure*)

Create a new *Ligand* object. The criteria to qualify for a ligand are defined by the global config (`DefaultConfig["ligand"]`).

Whether a ligand molecule is created is subject to several checks:

```
#. It has a certain number of atoms.
#. It has a certain number of contacts with the polymer.
#. It contacts a certain number of residues in the polymer.
#. Its atoms span a single chain.
```

If a ligand doesn't pass any of these checks, the function returns `None`.

Parameters

- **m_lig** (*npt.NDArray[np.bool_]*) – A boolean mask pointing to putative ligand atoms.
- **m_pol** (*npt.NDArray[np.bool_]*) – A boolean mask pointing to polymer atoms that supposedly contact ligand atoms.
- **structure** (GenericStructure) – A parent structure to which the masks can be applied.

Returns
An instantiated ligand or `None` if the checks were not passed.

Return type
Ligand | None

lXtractor.core.pocket module

The module defines *Pocket*, representing an arbitrarily defined binding pocket.

class lXtractor.core.pocket.**Pocket**(*definition*, *name='Pocket'*)

Bases: `object`

A binding pocket.

The pocket is defined via a single string following a particular syntax (a definition), such that, when applied to a ligand using *is_connected()*, the latter outputs `True` if ligand is connected. Consequently, it is tightly bound to *lXtractor.core.ligand.Ligand*. Namely, the definition relies on two matrices:

1. "c" = *lXtractor.core.ligand.Ligand.contact_mask* (boolean mask)
2. "d" = *lXtractor.core.ligand.Ligand.dist* (distances)

The definition is a combination of statements. Each statement involves the selection consisting of a matrix ("c" or "d"), residue positions, and residue atom names, formatted as:

```
{matrix-prefix}:[pos]:[atom_names] {sign} {number}
```

where `[pos]` and `[atom_names]` can be comma-separated lists, `sign` is` a comparison operator, and a `number` (`int` or `float`) is what to compare to. For instance, selection `c:1:CA,CB == 2` translates into "must have exactly two contacts with atoms "CA" and "CB" at position 1. See more examples below.

Comparison meaning depends on the matrix type used: "c" or "d".

In the former case, `>= x` means "at least x contacts". In the latter case, "`<=` x" means "have distance below x".

In the case of the "d" matrix, applying selection and comparison will result in a vector of `bool` bool values, requiring an aggregation. Two aggregation types are supported: "da" (any) and "daa" (all).

In the case of the "c" matrix, possible matrix prefixes are "c" and "cs". They have very different meanings! In the former case, the statements compares the total number of contacts when the selection is applied. In the latter case, the statement will select residues **separately** and, for each residue, decide whether the selected atoms form enough contact to extrapolate towards the full residue and mark it as "contacting" (controlled via *min_contacts*). These decisions are summed across each residue and this sum is compared to the number in the statement. See the example below.

Finally, statements can be bracketed and combined by boolean operators "AND" and "OR" (which one can abbreviate by "&" and "|").

Examples:

At least two contacts with any atom of residues 1 and 5:

```
c:1,5:any >= 2
```

Note that the above is a "cumulative" statement, i.e., it is applied to both residues at the same time. Thus, if a residue 1 has two atoms contacting a ligand while a residue 2 has none, this will still evaluate to `True`. The following definition will ensure that each residue has at least two contacts:

```
c:1:any >= 2 & c:2:any >= 2
```

In contrast, the following statement will translate "among residues 1, 2, and 3, there are at least two "contacting" residues:

```
cs:1,2,3:any >= 2
```

Any atoms farther than 10A from alpha-carbons of positions 1 and 10:

```
da:1,10:CA > 10
```

Any atoms with at least two contacts with any atoms at position 1 or all CA atoms closer than 6A of positions 2 and 3:

```
c:1:any >= 2 | daa:2,3:CA < 6
```

CA or CB atoms with a contact at position 1 but not 2, while position 3 has any atoms below 10A threshold:

```
c:1:CA,CB >= 1 & c:2:CA,CB == 0 & da:3:any <= 10
```

Contact with positions 1 and 2 or positions 3 and 4:

```
(c:1:any >= 1 & c:2:any >= 1) | (c:3:any >= 1 & c:4:any >= 1)
```

See also:

translate_definition().

__init__(*definition*, *name='Pocket'*)

is_connected(*ligand*, *mapping=None*, ***kwargs*)

Check whether a ligand is connected.

Parameters

- **ligand** (Ligand) – An arbitrary ligand.
- **mapping** (*dict[int, int] | None*) – A mapping to the ligand's parent structure numbering.
- **kwargs** – Passed to *translate_definition()*.

Returns

`True` if the ligand is bound within the pocket and `False` otherwise.

Return type

bool

definition

name

lXtractor.core.pocket.**make_sel**(*pos*, *atoms*)

Make a selection string from positions and atoms.

```
>>> make_sel(1, 'any')
'(a.res_id == 1)'
>>> make_sel([1, 2], 'CA,CB')
"np.isin(a.res_id, [1, 2]) & np.isin(a.atom_name, ['CA', 'CB'])"
```

Parameters

- **pos** (*int | Sequence[int]*) –
- **atoms** (*str*) –

Returns

Return type
str

lXtractor.core.pocket.**translate_definition**(*definition*, *mapping=None*, ***, *skip_unmapped=False*, *min_contacts=1*)

Translates the *Pocket.definition* into a series of statements, such that, when applied to ligand matrices, evaluate to bool.

```
>>> translate_definition("c:1:any > 1")
'(c[np.isin(a.res_id, [1])].sum() > 1)'
>>> translate_definition("da:1,2:CA,CZ <= 6")
"(d[np.isin(a.res_id, [1, 2]) & np.isin(a.atom_name, ['CA', 'CZ'])] <= 6).any()"
>>> translate_definition("daa:1,2:any > 2", {1: 10}, skip_unmapped=True)
'(d[np.isin(a.res_id, [10])] > 2).all()'
>>> translate_definition("cs:1,2:any > 2")
'sum([c[(a.res_id == 1)].sum() >= 1, c[(a.res_id == 2)].sum() >= 1]) > 2'
```

Warning: skip_unmapped=True may change the pocket's definition and lead to undesired conclusions. Caution advised!

Parameters

- **definition** (*str*) – A string definition of a *Pocket*.
- **mapping** (*dict[int, int] | None*) – An optional mapping from the definition's numbering to a structure's numbering.
- **skip_unmapped** (*bool*) – If the *mapping* is provided and some position is left unmapped, skip this position.
- **min_contacts** (*int*) – If prefix is "cs", use this threshold to determine a minimum number of residue contacts required to consider it bound.

Returns
A new string with statements of the provided definition translated into a numpy syntax.

Return type
str

lXtractor.core.segment module

Module defines a segment object serving as base class for sequences in lXtractor.

class lXtractor.core.segment.**Segment**(*start*, *end*, *name='S'*, *seqs=None*, *parent=None*, *children=None*, *meta=None*, *variables=None*)

Bases: Sequence[*NamedTupleT*]

An arbitrary segment with inclusive boundaries containing arbitrary number of sequences.

Sequences themselves may be retrieved via [] syntax:

```
>>> s = Segment(1, 10, 'S', seqs={'X': list(range(10))})
>>> s.id == 'S|1-10'
True
>>> s['X'] == list(range(10))
True
>>> 'X' in s
True
```

One can use the same syntax to check if a Segment contains certain index:

```
>>> 1 in s and 10 in s and not 11 in s
True
```

Iteration over the segment yields it's items:

```
>>> next(iter(s))
Item(i=1, X=0)
```

One can just get the same item by explicit index:

```
>>> s[1]
Item(i=1, X=0)
```

Slicing returns an iterable slice object:

```
>>> list(s[1:2])
[Item(i=1, X=0), Item(i=2, X=1)]
```

One can add a new sequence in two ways.

1) using a method:

```
>>> s.add_seq('Y', tuple(range(10, 20)))
>>> 'Y' in s
True
```

2) using [] syntax:

```
>>> s['Y'] = tuple(range(10, 20))
>>> 'Y' in s
True
```

Note that using the first method, if s already contains Y, this will cause an exception. To overwrite a sequence with the same name, please use explicit [] syntax.

Additionally, one can offset Segment indices using >>/<< syntax. This operation mutates original Segment!

```
>>> s >> 1
S|2-11
>>> 11 in s
True
```

__init__(*start*, *end*, *name='S'*, *seqs=None*, *parent=None*, *children=None*, *meta=None*, *variables=None*)

Parameters

- **start** (*int*) – Start coordinate.
- **end** (*int*) – End coordinate.
- **name** (*str*) – The name of the segment. Name with start and end coordinates should uniquely specify the segment. They are used to dynamically construct *id()*.
- **seqs** (*dict[str, abc.Sequence[t.Any]] | None*) – A dictionary name => sequence, where sequence is some sequence (preferably mutable) bounded by segment. Name of a sequence must be "simple", i.e., convertable to a field of a namedtuple.
- **parent** (*t.Self | None*) – Parental segment bounding this instance, typically obtained via *sub()* or *sub_by()* methods.
- **children** (*abc.MutableSequence[t.Self] | None*) – A mapping name => *Segment* with child segments bounded by this instance.
- **meta** (*dict[str, t.Any] | None*) – A dictionary with any meta-information str() => str() since reading/writing *meta* to disc will inevitably convert values to strings.
- **variables** (*Variables | None*) – A collection of variables calculated or staged for calculation for this segment.

add_seq(*name*, *seq*)

Add sequence to this segment.

Parameters

- **name** (*str*) – Sequence's name. Should be convertible to the namedtuple's field.
- **seq** (*Sequence[Any]*) – A sequence with arbitrary elements and the length of a segment.

Returns

returns nothing. This operation mutates *attr: `seqs*.

Raises

ValueError – If the *name* is reserved by another segment.

Return type

None

append(*other*, *filler=<function Segment.<lambda>>*, *joiner=<built-in function add>*)

Append another segment to this one.

The encompassed sequences will be merged together by *joiner*. If a sequence is missing in this segment or *other*, *filler* will create a sequence with filled values. The sequences will be deep-copied before merge.

```
>>> a = Segment(1, 3, "A", seqs={"A": "AAA"})
>>> b = Segment(1, 2, "B", seqs={"B": "BB"})
>>> c = a.append(b, filler=lambda x: '*' * x)
>>> c.id
'A|1-5'
>>> c['A']
'AAA**'
>>> c['B']
'***BB'
```

Note that the same can be achieved via | operator:

```
>>> a | b == a.append(b, filler=lambda x: '*' * x)
True
```

This will use "*" filler for `str`-type sequences and `None` for the rest and use the default *joiner* for joining them.

Note: Appending to an empty segment will return *other*. Appending an empty segment will return this segment.

Warning: Appending creates a new segment and removes associated parent and metadata

Parameters

- **other** (*t.Self*) – Another arbitrary segment.
- **filler** (*_Filler | abc.Mapping[str, _Filler]*) – A callable accepting the positive integer and returning a filled in a sequence or a `dict` mapping sequence names to such callables.
- **joiner** (*_Joiner | abc.Mapping[str, _Joiner]*) – A callable accepting two sequences and returning a merged sequence or a `dict` mapping sequence names to such callables.

Returns
A new segment with the same name as this segment, extended by *other*.

Return type
t.Self

bounded_by(*other*)

Check whether this segment is bounded by other.

```
self:   +----+
other: +------+
=> True
```

:param other; Another segment.

Return type
bool

bounds(*other*)

Check if this segment bounds other.

```
self: +-------+
other:  +----+
=> True
```

:param other; Another segment.

Return type
bool

id_strip_parents()

Returns
An identifier of this segment without parent information.

insert(*other*, *i*, ***kwargs*)

Insert a segment into this one.

The function splits this segment into two parts at the provided index and insert *other* between them via *append()*. The latter handles common/unique sequences via *filler* and *joiner* arguments, which can be passed here as keyword arguments.

Note: Inserting an empty segment returns this instance. Inserting a segment at the *end()* appends *other*.

Warning: Inserting creates a new segment and removes associated parent and metadata

Parameters

- **other** (*t.Self*) – Another segment to insert.
- **i** (*int*) – Index to insert at. The insertion will be performed after *i*.
- **kwargs** – Passed to *append()*.

Returns
A new segment with inserted *other*.

Raises
IndexError – If attempting to insert at an invalid index. Only indices `start < i <= end` are valid.

Return type
t.Self

overlap(*start*, *end*)

Create new segment from the current instance using overlapping boundaries.

Parameters

- **start** (*int*) – Starting coordinate.
- **end** (*int*) – Ending coordinate.

Returns
New overlapping segment with `data` and *name*

Return type
t.Self

overlap_with(*other*, *deep_copy=True*, *handle_mode='merge'*, *sep='&'*)

Overlap this segment with other over common indices.

```
self: +---------+
other:    +-------+
=>:       +-----+
```

Parameters

- **other** (Segment) – other *Segment* instance.
- **deep_copy** (*bool*) – deepcopy seqs to avoid side effects.
- **handle_mode** (*str*) – When the child overlapping segment is created, this parameter defines how *name* and *meta* are handled. The following values are possible:
  - ”merge”: merge meta and name from *self* and *other*
  - ”self”: the current instance provides both attributes
  - ”other”: *other* provides both attributes
- **sep** (*str*) – If *handle_mode* == “merge”, the new name is created by joining names of *self* and *other* using this separator.

Returns
New segment instance with inherited name and meta.

Return type
t.Self

overlaps(*other*)

Check whether a segment overlaps with the other segment. Use *overlap_with()* to produce an overlapping child *Segment*.

Parameters
other (Segment) – other *Segment* instance.

Returns
`True` if segments overlap and `False` otherwise.

Return type
bool

remove_seq(*name*)

Remove sequence from this segment.

Parameters
name (*str*) – Sequence’s name. If doesn’t exist in this segment, nothing happens.

sub(*start*, *end*, ***kwargs*)

Subset current segment using provided boundaries. Will create a new segment and call *sub_by()*.

Parameters

- **start** (*int*) – new start.
- **end** (*int*) – new end.
- **kwargs** – passed to *overlap_with()*

Return type
t.Self

sub_by(*other*, ***kwargs*)

A specialized version of *overlap_with()* used in cases where *other* is assumed to be a part of the current segment (hence, a subsegment).

Parameters

- **other** (Segment) – Some other segment contained within the (*start*, *end*) boundaries.
- **kwargs** – Passed to *overlap_with()*.

Returns
A new *Segment* object with boundaries of *other*. See *overlap_with()* on how to handle segments' names and data.

Raises
NoOverlap – If *other*'s boundaries lie outside the existing *start*, *end*.

Return type
t.Self

children

property end: int

Returns
A Segment's end coordinate.

property id: str

Returns

Unique segment's identifier encapsulating name, boundaries and parents of a segment if it was spawned from another *Segment* instance. For example:

```
S|1-2<-(P|1-10)
```

would specify a segment *S* with boundaries `[1, 2]` descended from *P*.

property is_empty: bool

Returns
`True` if the segment is empty. Emptiness is a special case, in which *Segment* has `start == end == 0`.

property is_singleton: bool

Returns
`True` if the segment contains a single element. In this special case, `start == end`.

property item_type: _Item

A factory to make an *Item* namedtuple object encapsulating sequence names contained within this instance. The first field is reserved for "i" – an index. :return: *Item* namedtuple object.

meta: dict[str, t.Any]

property name: str

property parent: t.Self | None

property seq_names: list[str]

Returns
A list of sequence names this segment entails.

property start: int

Returns
A Segment's start coordinate.

variables: *Variables*

lXtractor.core.segment.**do_overlap**(*segments*)

Check if any pair of segments overlap.

Parameters
segments (*Iterable[Segment]*) – an iterable with at least two segments.

Returns
`True` if there are overlapping segments, `False` otherwise.

Return type
bool

lXtractor.core.segment.**map_segment_numbering**(*segments_from*, *segments_to*)

Create a continuous mapping between the numberings of two segment collections. They must contain the same number of equal length non-overlapping segments. Segments in the *segments_from* collection are considered to span a continuous sequence, possibly interrupted due to discontinuities in a sequence represented by *segments_to*'s segments. Hence, the segments in *segments_from* form continuous numbering over which numberings of *segments_to* segments are joined.

Parameters

- **segments_from** (*Sequence[Segment]*) – A sequence of segments to map from.
- **segments_to** (*Sequence[Segment]*) – A sequence of segments to map to.

Returns
An iterable over (key, value) pairs. Keys correspond to numberings of the *segments_from*, values – to numberings of *segments_to*.

Return type
Iterator[tuple[int, int | None]]

lXtractor.core.segment.**resolve_overlaps**(*segments*, *value_fn=<built-in function len>*, *max_it=None*, *verbose=False*)

Eliminate overlapping segments.

Convert segments into and undirected graph (see *segments2graph()*). Iterate over connected components. If a component has only a single node (no overlaps§), yield it. Otherwise, consider all possible non-overlapping subsets of nodes. Find a subset such that the sum of the *value_fn* over the segments is maximized and yield nodes from it.

Parameters

- **segments** (*Iterable[Segment]*) – A collection of possibly overlapping segments.
- **value_fn** (*Callable[[Segment], float]*) – A function accepting the segment and returning its value.
- **max_it** (*int | None*) – The maximum number of subsets to consider when resolving a group of overlapping segments.

- **verbose** (*bool*) – Progress bar and general info.

Returns
A collection of non-overlapping segments with maximum cumulative value. Note that the optimal solution is guaranteed iff the number of possible subsets for an overlapping group does not exceed *max_it*.

Return type
Generator[Segment, None, None]

lXtractor.core.segment.**segments2graph**(*segments*)

Convert segments to an undirected graph such that segments are nodes and edges are drawn between overlapping segments.

Parameters
segments (*Iterable[Segment]*) – an iterable with segments objects.

Returns
an undirected graph.

Return type
Graph

lXtractor.core.structure module

Module defines basic interfaces to interact with macromolecular structures.

class lXtractor.core.structure.**CarbohydrateStructure**(*array*, *structure_id*, *ligands=True*, *atom_marks=None*, *graph=None*)

Bases: *GenericStructure*

A structure type where primary polymer is carbohydrate.

See also:

GenericStructure for general-purpose documentation.

__init__(*array*, *structure_id*, *ligands=True*, *atom_marks=None*, *graph=None*)

Parameters

- **array** (*AtomArray*) – Atom array object.
- **name** – ID of a structure in *array*.
- **ligands** (*bool | list[Ligand]*) – A list of ligands or flag indicating to extract ligands during initialization.

class lXtractor.core.structure.**GenericStructure**(*array*, *name*, *ligands=None*, *atom_marks=None*, *graph=None*)

Bases: object

A generic macromolecular structure with possibly many chains holding a single biotite.structure.AtomArray instance.

This object is a core data structure in *lXtractor* for structural data.

The object is considered immutable: atoms of a structure can't change their location or properties, as well as other protected attributes.

While atoms are stored as `biotite.structure.AtomArray`, *GenericStructure* defines additional annotations for each atom and operations crucial for other objects such as `lXtractor.core.chain.ChainStructure`.

Upon initialization, atom array attains graph representation (*graph()*) using *lXtractor.util.structure.to_graph()* function. Using this representation, atom annotations are attained via :func``mark_atoms_g`. These annotations can be accessed via *atom_marks()*. For convenience, boolean masks are stored and can be applied to the *array()* as follows:

```
# Assume ``s`` is a :class:`GenericStructure` object.
s[s.mask.`mask_name`]
```

To view available mask names, see *Masks*.

One of the most crucial annotations is the so-called "primary_polymer". These atoms serve as a frame of reference for all other atoms in a structure. The rest of the atoms are categorized as either ligand or solvent. Sometimes the annotation process fails to identify certain atoms. In such cases, a warning is logged. To view uncategorized atoms, one can use the following mask:

```
s[s.mask.unk]
```

Note: Using `__getitem__(item)` like in `s[s.mask.unk` will return an atom array. Use *subset()* to obtain a new generic structure or initialize a new ``GenericStructure(s[s.mask.unk] instance; it will be equivalent.

Methods `__repr__` and `__str__` output a string in the format: `{_name}:{polymer_chain_ids};{ligand_chain_ids}|{altloc_ids}` where `*ids` are ","-separated.

__init__(*array*, *name*, *ligands=None*, *atom_marks=None*, *graph=None*)

Parameters

- **array** (*AtomArray*) – Atom array object.
- **name** (*str*) – ID of a structure in *array*.
- **ligands** (*Sequence[Ligand] | None*) – A list of ligands or flag indicating to extract ligands during initialization.

extract_positions(*pos*, *chain_ids=None*, ***kwargs*)

Extract specific positions from this structure.

Parameters

- **pos** (*abc.Sequence[int]*) – A sequence of positions (res_id) to extract.
- **chain_ids** (*abc.Sequence[str] | str | None*) – Optionally, a single chain ID or a sequence of such.
- **kwargs** – Passed to *subset()*.

Returns

A new instance with extracted residues.

Return type

t.Self

extract_segment(*start*, *end*, *chain_id*, ***kwargs*)

Create a sub-structure encompassing some continuous segment bounded by existing position boundaries.

Parameters

- **start** (*int*) – Residue number to start from (inclusive).
- **end** (*int*) – Residue number to stop at (inclusive).
- **chain_id** (*str*) – Chain to extract a segment from.
- **kwargs** – Passed to *subset()*.

Returns
A new Generic structure with residues in `[start, end]`.

Return type
t.Self

`get_sequence()`

Returns
A generator over tuples, where each residue is described by: (1) one-letter code, (2) three-letter code, (3) residue number.

Return type
Generator[tuple[str, str, int]]

`classmethod make_empty(structure_id='XXXX')`

Parameters
structure_id (*str*) – (Optional) ID of the created array.

Returns
An instance with empty *array()*.

Return type
t.Self

`classmethod read(inp, path2id=<function GenericStructure.<lambda>>, structure_id='XXXX', altloc=False, **kwargs)`

Parse the atom array from the provided input and wrap it into the *GenericStructure* object.

See also:

lXtractor.util.structure.load_structure()

Note: If *inp* is not a `Path`, `kwargs` must contain the correct `fmt` (e.g., `fmt=cif`).

Parameters

- **inp** (*IOBase | Path | str | bytes*) – Path to a structure in supported format.
- **path2id** (*abc.Callable[[Path], str]*) – A callable obtaining a PDB ID from the file path. By default, it's a `Path.stem`.
- **structure_id** (*str*) – A structure unique identifier (e.g., PDB ID). If not provided and the input is `Path`, will use `path2id` to infer the ID. Otherwise, will use a constant placeholder.
- **altloc** (*bool | str*) – Parse alternative locations and populate `array.altloc_id` attribute.
- **kwargs** – Passed to `load_structure`.

Returns
Parsed structure.

Return type
t.Self

`rm_solvent`(*copy=False*)

Parameters
copy (*bool*) – Copy the resulting substructure.

Returns
A substructure with solvent molecules removed.

`split_altloc`(***kwargs*)

Split into substructures based on altloc IDs. Atoms missing altloc annotations are distributed into every substructure. Thus, even if a structure contains a single atom having altlocs (say, A and B), this method will produce two substructed identical except for this atom.

Note: If *array()* does not specify any altloc ID, the method yields `self`.

Parameters
kwargs – Passed to *subset()*.

Returns
An iterator over objects of the same type initialized by atoms having altloc annotations.

Return type
abc.Iterator[t.Self]

`split_chains`(*polymer=False*, ***kwargs*)

Split into separate chains. Splitting is done using `biotite.structure.get_chain_starts()`.

Note: Preserved ligands may have a different `chain_id`.

Note: If there is a single chain, this method will return `self`.

Parameters

- **polymer** (*bool*) – Use only primary polymer chains for splitting.
- **kwargs** – Passed to *subset()*.

Returns
An iterable over chains found in *array*.

Return type
abc.Iterator[t.Self]

`subset`(*mask*, *ligands=True*, *reinit_ligands=False*, *copy=False*)

Create a sub-structure potentially preserving connected *ligands()*.

Warning: If `DefaultConfig["structure"]["primary_pol_type"]` is set to auto, and *mask* points to a polymer that is shorter than some existing ligand polymer, this ligand polymer will become a primary polymer in the substructure.

Parameters

- **mask** (*np.ndarray*) – Boolean mask, `True` for atoms in *array()*, used to create a sub-structure.
- **ligands** (*bool*) – Keep ligands that are connected to atoms specified by *mask*.
- **reinit_ligands** (*bool*) – Reinitialize ligands upon creating a sub-structure, rather than filtering existing ligands connected to atoms specified by *mask*. Takes precedence over the *ligands* option. This option is used in *split_altloc()*.
- **copy** (*bool*) – Copy the atom array resulting from subsetting the original one.

Returns

A new instance with atoms defined by *mask* and connected ligands.

Return type

t.Self

superpose(*other*, *res_id_self=None*, *res_id_other=None*, *atom_names_self=None*, *atom_names_other=None*, *mask_self=None*, *mask_other=None*)

Superpose other structure to this one. Arguments to this function all serve a single purpose: to correctly subset both structures so the resulting selections have the same number of atoms.

The subsetting achieved either by specifying residue numbers and atom names or by supplying a binary mask of the same length as the number of atoms in the structure.

Parameters

- **other** (*GenericStructure | AtomArray*) – Other *GenericStructure* or atom array.
- **res_id_self** (*Iterable[int] | None*) – Residue numbers to select in this structure.
- **res_id_other** (*Iterable[int] | None*) – Residue numbers to select in other structure.
- **atom_names_self** (*Iterable[Sequence[str]] | Sequence[str] | None*) – Atom names to select in this structure given either per-residue or as a single sequence broadcasted to selected residues.
- **atom_names_other** (*Iterable[Sequence[str]] | Sequence[str] | None*) – Same as *self*.
- **mask_self** (*ndarray | None*) – Binary mask to select atoms. Takes precedence over other selection arguments.
- **mask_other** (*ndarray | None*) – Same as *self*.

Returns

A tuple of (1) an *other* structure superposed onto this one, (2) an RMSD of the superposition, and (3) a transformation that had been used with `biotite.structure.superimpose_apply()`.

Return type

tuple[GenericStructure, float, tuple[*ndarray*, *ndarray*, *ndarray*]]

write(*path*, *atom_marks=True*, *graph=True*)

Save this structure to a file. The format is automatically determined from the given path.

Additional files are saved using the same filename alongside the structure file. The filename will resolve to "structure" in all the following cases and result in "structure.npy" and "structure.json" files saved to the same dir:

```
path="/path/to/structure.pdb"
path="/path/to/structure.mmtf.gz"
path="/path/to/structure.with.many.dots.pdb.gz"
```

See also:

lXtractor.util.structure.save_structure().

Parameters

- **path** (*PathLike | str*) – A path or a path-like object compatible with `open()`. Must not point to an existing directory. Must provide the structure format as an extension.
- **atom_marks** (*bool*) – Save an array of atom marks in the *npy* format.
- **graph** (*bool*) – Save molecular connectivity graph in the *json* format.

Returns
Path to the saved structure if writing was successful.

Return type
Path

property altloc_ids: list[str]

Returns
A sorted list of altloc IDs. If none found, will output `[""]`.

property array: AtomArray

Returns
Atom array object.

property atom_marks: ndarray[Any, dtype[int64]]

Returns
An array of *lXtractor.core.config.AtomMark* marks, categorizing each atom in this structure.

property chain_ids: list[str]

Returns
A list of chain IDs this structure encompasses.

property chain_ids_ligand: list[str]

Returns
A set of ligand chain IDs.

property chain_ids_polymer: list[str]

Returns
A list of polymer chain IDs.

property graph: PyGraph

Returns
A structure's graph representation.

property id: str

Returns
An identifier of this structure. It's composed once upon initialization and has the following format: `{_name}:{polymer_chain_ids};{ligand_chain_ids}|{altloc_ids}`. It should uniquely identify a structure, i.e., one should expect two structures with the same ID to be identical.

property is_empty: bool

Returns
`True` if the *array()* is empty.

property is_empty_polymer: bool

Check if there are any polymer atoms.

Returns
`True` if there are >=1 polymer atoms and `False` otherwise.

property is_singleton: bool

Returns
`True` if the structure contains a single residue.

property ligands: tuple[*Ligand*, ...]

Returns
A list of ligands.

property mask: *Masks*

property name: str

Returns
A name of the structure.

class lXtractor.core.structure.**Masks**(*primary_polymer: 'npt.NDArray[np.bool_]'*, *primary_polymer_ptm: 'npt.NDArray[np.bool_]'*, *primary_polymer_modified: 'npt.NDArray[np.bool_]'*, *solvent: 'npt.NDArray[np.bool_]'*, *ligand: 'npt.NDArray[np.bool_]'*, *ligand_covalent: 'npt.NDArray[np.bool_]'*, *ligand_poly: 'npt.NDArray[np.bool_]'*, *ligand_nonpoly: 'npt.NDArray[np.bool_]'*, *ligand_pep: 'npt.NDArray[np.bool_]'*, *ligand_nuc: 'npt.NDArray[np.bool_]'*, *ligand_carb: 'npt.NDArray[np.bool_]'*, *unk: 'npt.NDArray[np.bool_]'*)

Bases: `object`

__init__(*primary_polymer*, *primary_polymer_ptm*, *primary_polymer_modified*, *solvent*, *ligand*, *ligand_covalent*, *ligand_poly*, *ligand_nonpoly*, *ligand_pep*, *ligand_nuc*, *ligand_carb*, *unk*)

ligand: ndarray[Any, dtype[bool_]]

ligand_carb: ndarray[Any, dtype[bool_]]

ligand_covalent: ndarray[Any, dtype[bool_]]

ligand_nonpoly: ndarray[Any, dtype[bool_]]

ligand_nuc: ndarray[Any, dtype[bool_]]

ligand_pep: ndarray[Any, dtype[bool_]]

ligand_poly: ndarray[Any, dtype[bool_]]

primary_polymer: ndarray[Any, dtype[bool_]]

primary_polymer_modified: ndarray[Any, dtype[bool_]]

primary_polymer_ptm: ndarray[Any, dtype[bool_]]

solvent: ndarray[Any, dtype[bool_]]

unk: ndarray[Any, dtype[bool_]]

class lXtractor.core.structure.**NucleotideStructure**(*array*, *structure_id*, *ligands=True*, *atom_marks=None*, *graph=None*)

Bases: *GenericStructure*

A structure type where primary polymer is nucleotide.

See also:

GenericStructure for general-purpose documentation.

__init__(*array*, *structure_id*, *ligands=True*, *atom_marks=None*, *graph=None*)

Parameters

- **array** (*AtomArray*) – Atom array object.
- **name** – ID of a structure in *array*.
- **ligands** (*bool | list[Ligand]*) – A list of ligands or flag indicating to extract ligands during initialization.

class lXtractor.core.structure.**ProteinStructure**(*array*, *structure_id*, *ligands=True*, *atom_marks=None*, *graph=None*)

Bases: *GenericStructure*

A structure type where primary polymer is peptide.

See also:

GenericStructure for general-purpose documentation.

__init__(*array*, *structure_id*, *ligands=True*, *atom_marks=None*, *graph=None*)

Parameters

- **array** (*AtomArray*) – Atom array object.
- **name** – ID of a structure in *array*.
- **ligands** (*bool | list[Ligand]*) – A list of ligands or flag indicating to extract ligands during initialization.

lXtractor.core.structure.**mark_atoms**(*structure*)

Mark each atom in structure according to *lXtractor.core.config.AtomMark*.

This function is used upon initializing *GenericStructure* and its subclasses, storing the output under *GenericStructure.atom_marks*.

Parameters
structure (GenericStructure) – An arbitrary structure.

Returns
An array of atom marks (equivalently, classes or types).

Return type
tuple[*ndarray*[*Any*, *dtype*[*int64*]], list[Ligand]]

lXtractor.core.structure.**mark_atoms_g**(*s*, *single_poly_chain=False*)

Mark structure atoms based on a molecular graph's representation by of the *lXtractor.core.config.AtomMark* categories.

Atoms are classified into five categories:

```
#. primary polymer: corresponds to ``PEP``, ``NUC`` or ``CARB``
categories.
#. solvent: ``SOLVENT``.
#. non polymer ligand: ``LIGAND``.
#. polymer ligand: A combination of ``LIGAND`` with one of the primary
polymer types, eg. ``AtomMark.LIGAND | AtomMark.NUC``.
#. unknown: ``UNK`` for atoms that couldn't be categorized.
```

The classification process depends on groups of atoms forming covalent bonds with each other, or connected components in the molecular graph representation. Each such component is assessed separately and its atoms are classified as polymer, ligand, or solvent. If the primary polymer is set to "auto" in config (`DefaultConfig["structure"]["primary_pol_type"]`), the polymer with the largest number of monomers will be selected. The rest of the polymers will become polymer ligands: special kind of ligand that can have multiple residues. See `lXtractore.core.ligand.Ligand` for details.

Parameters

- **s** (GenericStructure) –
- **single_poly_chain** (*bool*) –

Returns

Return type
(npt.NDArray[np.int_], str, list[*Ligand*])

3.1.2 lXtractor.chain package

lXtractor.chain.base module

lXtractor.chain.base.**is_chain_type**(*s*)

Return type
t.TypeGuard[CTU]

lXtractor.chain.base.**is_chain_type_iterable**(*s*)

Return type
t.TypeGuard[abc.Iterable[*Chain*] | abc.Iterable[*ChainSequence*] | abc.Iterable[*ChainStructure*]]

lXtractor.chain.base.**topo_iter**(*start_obj*, *iterator*)

Iterate over sequences in topological order.

```
>>> n = 1
>>> it = topo_iter(n, lambda x: (x + 1 for n in range(x)))
>>> next(it)
[2]
>>> next(it)
[3, 3]
```

Parameters

- **start_obj** (*T*) – Starting object.
- **iterator** (*Callable[[T], Iterable[T]]*) – A callable accepting a single argument of the same type as the *start_obj* and returning an iterator over objects with the same type, representing the next level.

Returns
A generator yielding lists of objects obtained using *iterator* and representing topological levels with the root in *start_obj*.

Return type
Generator[list[*T*], None, None]

lXtractor.chain.sequence module

class lXtractor.chain.sequence.**ChainSequence**(*start*, *end*, *name='S'*, *seqs=None*, *parent=None*, *children=None*, *meta=None*, *variables=None*)

Bases: *Segment*

A class representing polymeric sequence of a single entity (chain).

The sequences are stored internally as a dictionary *{seq_name => _seq}* and must all have the same length. Additionally, *seq_name* must be a valid field name: something one could use in namedtuples. If unsure, please use *lXtractor.util.misc.is_valid_field_name()* for testing.

A single gap-less primary sequence (*seq1()*) is mandatory during the initialization. We refer to the sequences other than *seq1()* as "maps." To view the standard sequence names supported by *ChainSequence*, use the `flied_names()` property.

The sequence can be a part of a larger one. The child-parent relationships are indicated via `parent` and attr:*children*, where the latter entails any sub-sequence. A preferable way to create subsequences is the *spawn_child()* method.

```
>>> seqs = {
...     'seq1': 'A' * 10,
...     'A': ['A', 'N', 'Y', 'T', 'H', 'I', 'N', 'G', '!', '?']
... }
>>> cs = ChainSequence(1, 10, 'CS', seqs=seqs)
>>> cs
CS|1-10
```

(continues on next page)

(continued from previous page)

```
>>> assert len(cs) == 10
>>> assert 'A' in cs and 'seq1' in cs
>>> assert cs.seq1 == 'A' * 10
```

apply_children(*fn*, *inplace=False*)

Apply some function to children.

Parameters

- **fn** (*ApplyT[ChainSequence]*) – A callable accepting and returning the chain sequence type instance.
- **inplace** (*bool*) – Apply to children in place. Otherwise, return a copy with only children transformed.

Returns

A chain sequence with transformed children.

Return type

t.Self

apply_to_map(*map_name*, *fn*, *inplace=False*, *preserve_children=False*, *apply_to_children=False*)

Apply some function to map/sequence in this chain sequence.

Parameters

- **map_name** (*str*) – Name of the internal sequence/map.
- **fn** (*ApplyT[abc.Sequence]*) – A function accepting and returning a sequence of the same length.
- **inplace** (*bool*) – Apply the operation to this object. Otherwise, create a copy with the transformed sequence.
- **preserve_children** (*bool*) – Preserve `children` of this instance in the transformed object. Passing `True` makes sense if the target sequence is mutable: the children's will be transformed naturally. In the target sequence is immutable, consider passing `True` with `apply_to_children=True`.
- **apply_to_children** (*bool*) – Recursively apply the same *fn* to a child tree starting from this instance. If passed, sets `preserve_children=True`: otherwise, one is at risk of removing all `children` in the child tree of the returned instance.

Returns

Return type

t.Self

as_chain(*transfer_children=True*, *structures=None*, ***kwargs*)

Convert this chain sequence to chain.

Note: Pass `add_to_children=True` to transfer *structure* to each child if `transfer_children=True`.

Parameters

- **transfer_children** (*bool*) – Transfer existing children.

- **structures** (*abc.Sequence[ChainStructure] | None*) – Add structures to the created chain.
- **kwargs** – Passed to `Chain.add_structure`

Returns

Return type
Chain

`as_df()`

Returns
The pandas DataFrame representation of the sequence where each column correspond to a sequence or map.

Return type
DataFrame

`as_np()`

Returns
The numpy representation of a sequence as matrix. This is a shortcut to *as_df()* and getting *df.values*.

Return type
ndarray

`coverage`(*map_names=None*, *save=True*, *prefix='cov'*)

Calculate maps' coverage, i.e., the number of non-empty elements.

Parameters

- **map_names** (*Sequence[str] | None*) – optionally, provide the sequence of map names to calculate the coverage for.
- **save** (*bool*) – save the results to `meta`
- **prefix** (*str*) – if *save* is `True`, format keys f"{prefix}_{name}" for the `meta` dictionary.

Returns

Return type
dict[str, float]

`fill`(*other*, *template*, *target*, *link_name*, *link_points_to*, *keep=True*, *target_new_name=None*, *empty_template=(None,)*, *empty_target=(None,)*, *transform=<function identity>*)

Fill-in a sequence in *other* using a template sequence from here.

As an example, consider two related sequences, `s` and `o`, mapped to the same reference numbering scheme `r`, which we'll denote as a "link sequence."

We would like to fill in "X" residues within `o` with residues from `s`. Let's first try this:

```
>>> s = ChainSequence.from_string('ABCD', r=[10, 11, 12, 13])
>>> o = ChainSequence.from_string('AABXDE', r=[9, 10, 11, 12, 13, 14])
>>> s.fill(o,'seq1','seq1','r','r')
['A', 'A', 'B', 'X', 'D', 'E']
```

In the example above, "X" was not replaced because it's not considered and "empty" target element requiring replacement. Below, we'll provide a tuple of possible empty values and pass a transform function that will join the result back into `str`.

```
>>> s.fill(o,'seq1','seq1','r','r',empty_target=('X', ),transform="".join)
'AABCDE'
>>> o['seq1_patched'] == 'AABCDE'
True
```

Parameters

- **other** (*t.Self*) – Some other chain sequence.
- **template** (*str*) – The name of the template sequence.
- **target** (*str*) – Target sequence name within *other* to patch.
- **link_name** (*str*) – Name of the map within *other* that links it with this sequence.
- **link_points_to** (*str | None*) – Name of the map within this chain sequence that corresponding to *link_name* within *other*. If `None`, it is assumed to be the same as *link_name*.
- **keep** (*bool*) – Keep patched sequence within *other*.
- **target_new_name** (*str | None*) – Name of the patched sequence to save within *other* if *keep* is `True`. If this or *target* names are "seq1", will use "seq1_patched" as *target_new_name* as this sequence is considered immutable by convention.
- **empty_target** (*tuple[t.Any, ...] | abc.Callable[[T], bool]*) – A tuple of element instances or a callable. If tuple, a *target* element will be replaced with the corresponding element from`template` if it's within this tuple. If callable, should accept an element of the target sequence and output `True` if it should be replaced with an element from the *template* and `False` otherwise.
- **empty_template** (*tuple[t.Any, ...] | abc.Callable[[T], bool]*) – Same as *empty_target* but applied to a template character, with reverse meaning for `True` and `False` of the *empty_target* param.
- **transform** (*abc.Callable[[list[T]], abc.Sequence[R]]*) – A function that transforms the result from one sequence to another.

Returns
A patched mapping/sequence after applying the *transform* function.

Return type
abc.Sequence[R]

filter_children(*pred*, *inplace=False*)

Filter children using some predicate.

Parameters

- **pred** (`FilterT[ChainSequence]`) – Some callable accepting chain sequence and returning bool.
- **inplace** (*bool*) – Filter `children` in place. Otherwise, return a copy with only children transformed.

Returns
A chain sequence with filtered children.

Return type
t.Self

classmethod from_df(*df*, *name='S'*, *meta=None*)

Init sequence from a data frame.

Parameters

- **df** (*Path | pd.DataFrame*) – Path to a tsv file or a pandas DataFrame.
- **name** (*str*) – Name of a new chain sequence.
- **meta** (*dict[str, t.Any] | None*) – Meta info of a new chain sequence.

Returns
Initialized chain sequence.

Return type
t.Self

classmethod from_file(*inp*, *reader=<function read_fasta>*, *start=None*, *end=None*, *name=None*, *meta=None*, ***kwargs*)

Initialize chain sequence from file.

Parameters

- **inp** (*Path | TextIOBase | Iterable[str]*) – Path to a file or file handle or iterable over file lines.
- **reader** (SeqReader) – A function to parse the sequence from *inp*.
- **start** (*int | None*) – Start coordinate of a sequence in a file. If not provided, assumed to be 1.
- **end** (*int | None*) – End coordinate of a sequence in a file. If not provided, will evaluate to the sequence's length.
- **name** (*str | None*) – Name of a sequence in *inp*. If not provided, will evaluate to a sequence's header.
- **meta** (*dict[str, Any] | None*) – Meta-info to add for the sequence.
- **kwargs** – Additional sequences other than *seq1* (as used during initialization via *_seq* attribute).

Returns
Initialized chain sequence.

Return type
ChainSequence

classmethod from_string(*s*, *start=None*, *end=None*, *name='S'*, *meta=None*, ***kwargs*)

Initialize chain sequence from string.

Parameters

- **s** (*str*) – String to init from.
- **start** (*int | None*) – Start coordinate (default=1).
- **end** (*int | None*) – End coordinate(default=len(s)).
- **name** (*str*) – Name of a new chain sequence.
- **meta** (*dict[str, Any] | None*) – Meta info of a new sequence.
- **kwargs** – Additional sequences other than *seq1* (as used during initialization via *_seq* attribute).

Returns
Initialized chain sequence.

Return type
ChainSequence

classmethod from_tuple(*inp*, *start=None*, *end=None*, *meta=None*, ***kwargs*)

get_closest(*key*, *value*, ***, *reverse=False*)

Find the closest item for which item.key >=/<= value. By default, the search starts from the sequence's beginning, and expands towards the end until the first element for which the retrieved *value* >= the provided *value*. If the *reverse* is `True`, the search direction is reversed, and the comparison operator becomes <=

```
>>> s = ChainSequence(1, 4, 'CS', seqs={'seq1': 'ABCD', 'X': [5, 6, 7, 8]})
>>> s.get_closest('seq1', 'D')
Item(i=4, seq1='D', X=8)
>>> s.get_closest('X', 0)
Item(i=1, seq1='A', X=5)
>>> assert s.get_closest('X', 0, reverse=True) is None
```

Parameters

- **key** (*str*) – map name.
- **value** (Ord) – map value. Must support comparison operators.
- **reverse** (*bool*) – reverse the sequence order and the comparison operator.

Returns
The first relevant item or *None* if no relevant items were found.

Return type
NamedTupleT | None

get_item(*key*, *value*)

Get a specific item. Same as *get_map()*, but uses *value* to retrieve the needed item immediately.

(!) Use it when a single item is needed. For multiple queries for the same sequence, please use *get_map()*.

```
>>> s = ChainSequence.from_string('ABC', name='CS')
>>> s.get_item('seq1', 'B').i
2
```

Parameters

- **key** (*str*) – map name.
- **value** (*Any*) – sequence value of the sequence under the *key* name.

Returns
an item correpsonding to the desired sequence element.

Return type
NamedTupleT

get_map(*key*, *to=None*, *rm_empty=False*)

Obtain the mapping of the form "key->item(seq_name=*,...)".

```
>>> s = ChainSequence.from_string('ABC', name='CS')
>>> s.get_map('i')
{1: Item(i=1, seq1='A'), 2: Item(i=2, seq1='B'), 3: Item(i=3, seq1='C')}
>>> s.get_map('seq1')
{'A': Item(i=1, seq1='A'), 'B': Item(i=2, seq1='B'), 'C': Item(i=3, seq1='C')}
>>> s.add_seq('S', [1, 2, np.nan])
>>> s.get_map('seq1', 'S', rm_empty=True)
{'A': 1, 'B': 2}
```

Parameters

- **key** (*str*) – A _seq name to map from.
- **to** (*str | None*) – A _seq name to map to.
- **rm_empty** (*bool*) – Remove empty keys and values. A numeric value is empty if it is of type NaN. A string value is empty if it is an empty string ("").

Returns

dict mapping key values to items.

Return type

dict[*Hashable*, *Any*]

iter_children()

Iterate over a child tree in topological order.

```
>>> s = ChainSequence(1, 10, 'CS', seqs={'seq1': 'A' * 10})
>>> ss = s.spawn_child(1, 5, 'CS_')
>>> sss = ss.spawn_child(1, 3, 'CS__')
>>> list(s.iter_children())
[[CS_|1-5<-(CS|1-10)], [CS__|1-3<-(CS_|1-5<-(CS|1-10))]]
```

Returns

a generator over child tree levels, starting from the `children` and expanding such attributes over *ChainSequence* instances within this attribute.

Return type

Generator[ChainList[ChainSequence], None, None]

classmethod make_empty(***kwargs*)

Returns

An empty chain sequence.

Return type

ChainSequence

map_boundaries(*start*, *end*, *map_name*, *closest=False*)

Map the provided boundaries onto sequence.

A convenient interface for common task where one wants to find sequence elements corresponding to arbitrary boundaries.

```
>>> s = ChainSequence.from_string('XXSEQXX', name='CS')
>>> s.add_seq('NCS', list(range(10, 17)))
>>> s.map_boundaries(1, 3, 'i')
(Item(i=1, seq1='X', NCS=10), Item(i=3, seq1='S', NCS=12))
>>> s.map_boundaries(5, 12, 'NCS', closest=True)
(Item(i=1, seq1='X', NCS=10), Item(i=3, seq1='S', NCS=12))
```

Parameters

- **start** (Ord) – Some orderable object.
- **end** (Ord) – Some orderable object.
- **map_name** (*str*) – Use this sequence to search for boundaries. It is assumed that `map_name in self is True`.
- **closest** (*bool*) – If true, instead of exact mapping, search for the closest elements.

Returns
a tuple with two items corresponding to mapped *start* and *end*.

Return type
tuple[NamedTupleT, NamedTupleT]

map_numbering(*other*, *align_method=<function mafft_align>*, *save=True*, *name='S'*, ***kwargs*)

Map the *numbering()*: of another sequence onto this one. For this, align primary sequences and relate their numbering.

```
>>> s = ChainSequence.from_string('XXSEQXX', name='CS')
>>> o = ChainSequence.from_string('SEQ', name='CSO')
>>> s.map_numbering(o)
[None, None, 1, 2, 3, None, None]
>>> assert 'map_CSO' in s
>>> a = Alignment([('CS1', 'XSEQX'), ('CS2', 'XXEQX')])
>>> s.map_numbering(a, name='map_aln')
[None, 1, 2, 3, 4, 5, None]
>>> assert 'map_aln' in s
```

Parameters

- **other** (*str | tuple[str, str]* | ChainSequence | Alignment) – another chain _seq.
- **align_method** (AlignMethod) – a method to use for alignment.
- **save** (*bool*) – save the numbering as a sequence.
- **name** (*str*) – a name to use if *save* is `True`.
- **kwargs** – passed to *func:map_pairs_numbering*.

Returns
a list of integers with `None` indicating gaps.

Return type
list[None | int]

match(*map_name1*, *map_name2*, *as_fraction=True*, *save=True*, *name='auto'*)

Parameters

- **map_name1** (*str*) – Mapping name 1.
- **map_name2** (*str*) – Mapping name 2.
- **as_fraction** (*bool*) – Divide by the total length.
- **save** (*bool*) – Save the result to `meta`.
- **name** (*str*) – Name of the saved metadata entry. If "auto", will derive from given map names.

Returns
The total number or a fraction of matching characters between maps.

Return type
float

patch(*other*, *numerator*, *link_name*, *link_points_to*, *diff=<built-in function sub>*, *num_filter=<function ChainSequence.<lambda>>*, ***kwargs*)

Patch the gaps in the provided sequence using this sequence as template.

The existence of a gap is judged by the *numerator* map that should point to a numeration scheme. If there are two consecutive *numerator* elements, for which *diff* returns value greater than one, this is considered a gap that could be filled in by a template.

To relate a potential gap to the template sequence, a link sequence must exist in the provided sequence, containing values referencing the template.

As an example, consider the template sequence "ABCDEG" and the sequence requiring patching "BDEG". Let `e` be the numbering of the "BDEG", `e=[1, 4, 5, 6]` and `r=[2, 4, 5, 6]` be a link map that points to the segment indices of the template.

```
>>> template = ChainSequence.from_string("ABCDEG", name='T')
>>> seq = ChainSequence.from_string("BDEG", name='P', e=[1,4,6,7], r=[2,4,5,6])
```

Observe that there is a numeration gap between `1` and `4`. The corresponding elements of `r` point to the template indices `2` an `4`. Thus, there is a gap that can be filled in by a portion of the template between `2` and `4`. Here, it turns out to be singleton sequence element "C" at position `3`. This segment will be inserted into the patched sequence:

```
>>> patched = template.patch(seq,'e','r','i')
>>> patched.id
'P|1-5'
>>> patched.seq1
'BCDEG'
```

Similar to *patch()*, the sequence elements missing in either of the sequences will be filled-in. Thus, what happens to the original numeration `e`?

```
>>> patched['e']
[1, None, 4, 6, 7]
```

On the other hand, the link sequence `r` can be successfully filled in by the template:

```
>>> patched['r']
[2, 3, 4, 5, 6]
```

Note: If this segment is empty or singleton, the *other* is returned unchanged.

Warning: This operation creates a new segment. The parents and metadata won't be transferred.

See also:

lXtractor.core.segment.Segment.insert() used to insert segments while patching.

Parameters

- **other** (*t.Self*) – A sequence to patch.
- **numerator** (*str*) – A map name in *other* containing numeration scheme the gaps will be inferred from.
- **link_name** (*str*) – A map name in *other* with values referencing some sequence in this instance.
- **link_points_to** (*str*) – A map name in this instance that the *link_name* refers to in *other*.
- **diff** (*abc.Callable[[T, T], int]*) – A callable accepting two *numerator* elements – higher and lower ones – and returning the number of elements between them. By default, a simple substraction is used.
- **num_filter** (*abc.Callable[[t.Any], bool]*) – An optional filter function to filter out elements in the *numerator* before splitting it into consecutive pairs. By default, this function will filter out any `None` values.
- **kwargs** – Additional keyword arguments passed to meth:*lXtractor.core.segment.Segment.insert*.

Returns
A new patched segment.

Return type
t.Self

classmethod read(*base_dir*, ***, *search_children=False*)

Initialize chain sequence from dump created using *write()*.

Parameters

- **base_dir** (*Path*) – A path to a dump dir.
- **search_children** (*bool*) – Recursively search for child segments and populate the `children`

Returns
Initialized chain sequence.

Return type
t.Self

relate(*other*, *map_name*, *link_name*, *link_points_to='i'*, *keep=True*, *map_name_in_other=None*)

Relate mapping from this sequence with *other* via some common "link" sequence.

The "link" sequence is a part of the *other* pointing to some sequence within this instance.

As an example, consider the case of transferring the mapping to alignment positions *aln_map*. To do this, the *other* must be mapped to some sequence within this instance – typically to canonical numbering – via some stored *map_canonical* sequence.

Thus, one would use ..code-block:: python

this.relate(
other, map_name=aln_map, link_name=map_canonical, link_name_points_to="i"

)

In the example below, we transfer *map_some* sequence from *s* to *o* via sequence *L* pointing to the primary sequence of *s*:

```
seq1    : A B C D   ---|
map_some: 9 8 7 6      | --> 9 8 None 6 (map transferred to `o`)
          | | | |      |
seq1    : X Y Z R      |
L       : A B X D   ---|
```

```
>>> s = ChainSequence.from_string('ABCD', name='CS')
>>> s.add_seq('map_some', [9, 8, 7, 6])
>>> o = ChainSequence.from_string('XYZR', name='XY')
>>> o.add_seq('L', ['A', 'B', 'X', 'D'])
>>> assert 'L' in o
>>> s.relate(o,map_name='map_some', link_name='L', link_points_to='seq1')
[9, 8, None, 6]
>>> assert o['map_some'] == [9, 8, None, 6]
```

Parameters

- **other** (*t.Self*) – An arbitrary chain sequence.
- **map_name** (*str*) – The name of the sequence to transfer.
- **link_name** (*str*) – The name of the "link" sequence that connects *self* and *other*.
- **link_points_to** (*str*) – Values within this instance the "link" sequence points to.
- **keep** (*bool*) – Store the obtained sequence within the *other*.
- **map_name_in_other** (*str | None*) – The name of the mapped sequence to store within the *other*. By default, the *map_name* is used.

Returns
The mapped sequence.

Return type
list[t.Any]

rename(*name*)

Rename this sequence by modifying the `name`.

Note: This is a mutable operation. Returning a copy of this sequence upon renaming will create two identical sequences with different IDs, which is discouraged.

Parameters
name (*str*) – New name.

Returns
The same sequence with a new name.

Return type
t.Self

spawn_child(*start*, *end*, *name=None*, *category=None*, ***, *map_from=None*, *map_closest=False*, *deep_copy=False*, *keep=True*)

Spawn the sub-sequence from the current instance.

Child sequence's boundaries must be within this sequence's boundaries.

Uses `Segment.sub()` method.

```
>>> s = ChainSequence(
...     1, 4, 'CS',
...     seqs={'seq1': 'ABCD', 'X': [5, 6, 7, 8]}
... )
>>> child1 = s.spawn_child(1, 3, 'Child1')
>>> assert child1.id in s.children
>>> s.children
[Child1|1-3<-(CS|1-4)]
```

Parameters

- **start** (*int*) – Start of the sub-sequence.
- **end** (*int*) – End of the sub-sequence.
- **name** (*str | None*) – Spawned child sequence's name.
- **category** (*str | None*) – Spawned child category. Any meaningful tag string that could be used later to group similar children.
- **map_from** (*str | None*) – Optionally, the map name the boundaries correspond to.
- **map_closest** (*bool*) – Map to closest *start*, *end* boundaries (see *map_boundaries()*).
- **deep_copy** (*bool*) – Deep copy inherited sequences.
- **keep** (*bool*) – Save child sequence within `children`.

Returns
Spawned sub-sequence.

Return type
ChainSequence

summary(*meta=True*, *children=False*)

Return type
DataFrame

write(*dest*, ***, *write_children=False*)

Dump this chain sequence. Creates *sequence.tsv* and *meta.tsv* in *base_dir* using *write_seq()* and *write_meta()*.

Parameters

- **dest** (*Path*) – Destination directory.
- **write_children** (*bool*) – Recursively write children.

Returns
Path to the directory where the files are written.

Return type
Path

`write_meta`(*path*, *sep='\t'*)

Write meta information as {key}{sep}{value} lines.

Parameters

- **path** (*Path*) – Write destination file.
- **sep** – Separator between key and value.

Returns
Nothing.

`write_seq`(*path*, *fields=None*, *sep='\t'*)

Write the sequence (and all its maps) as a table.

Parameters

- **path** (*Path*) – Write destination file.
- **fields** (*list[str] | None*) – Optionally, names of sequences to dump.
- **sep** (*str*) – Table separator. Please use the default to avoid ambiguities and keep readability.

Returns
Nothing.

`property categories: list[str]`

Returns
A list of categories associated with this object.

Categories are kept under "category" field in `meta` as a ","-separated list of strings. For instance, "domain,family_x".

`property fields: tuple[str, ...]`

Returns
Names of the currently stored sequences.

`property numbering: Sequence[int]`

Returns
the primary sequence's (*seq1()*) numbering.

`property seq: t.Self`

This property exists for functionality relying on the *.seq* attribute.

Returns
This object.

`property seq1: str`

Returns
the primary sequence.

property seq3: Sequence[str]

Returns
the three-letter codes of a primary sequence.

lXtractor.chain.sequence.**map_numbering_12many**(*obj_to_map*, *seqs*, *num_proc=1*, *verbose=False*, ***kwargs*)

Map numbering of a single sequence to many other sequences.

This function does not save mapped numberings.

See also:

ChainSequence.map_numbering().

Parameters

- **obj_to_map** (*str | tuple[str, str] | ChainSequence | Alignment*) – Object whose numbering should be mapped to *seqs*.
- **seqs** (*Iterable[ChainSequence]*) – Chain sequences to map the numbering to.
- **num_proc** (*int*) – A number of parallel processes to use.
- **verbose** (*bool*) – Output progress bar.
- **kwargs** – Passed to *lXtractor.util.misc.apply()*.

Returns
An iterator over the mapped numberings.

Return type
Iterator[list[int | None]]

lXtractor.chain.sequence.**map_numbering_many2many**(*objs_to_map*, *seq_groups*, *num_proc=1*, *verbose=False*, ***kwargs*)

Map numbering of each object *o* in *objs_to_map* to each sequence in each group of the *seq_groups*

```
o1 -> s1_1 s1_1 s1_3 ...
o2 -> s2_1 s2_1 s2_3 ...
          ...
```

This function does not save mapped numberings.

For a single object-group pair, it's the same as *map_numbering_12many()*. The benefit comes from parallelization of this functionality.

See also:

ChainSequence.map_numbering(). *map_numbering_12many()*

Parameters

- **objs_to_map** (*Sequence[str | tuple[str, str] | ChainSequence | Alignment]*) – An iterable over objects whose numbering to map.
- **seq_groups** (*Sequence[Sequence[ChainSequence]]*) – Group of objects to map numbering to.
- **num_proc** (*int*) – A number of processes to use.
- **verbose** (*bool*) – Output a progress bar.

- **kwargs** – Passed to *lXtractor.util.misc.apply()*.

Returns
An iterator over lists of lists with numeric mappings

Return type
Iterator[list[list[int | None]]]

```
[[s1_1 map, s1_2 map, ...]
 [s2_1 map, s2_2 map, ...]
           ...
 ]
```

lXtractor.chain.structure module

class lXtractor.chain.structure.**ChainStructure**(*structure*, *chain_id=None*, *structure_id=None*, *seq=None*, *parent=None*, *children=None*, *variables=None*)

Bases: `object`

A structure of a single chain.

Typical usage workflow:

1. **Use :meth:`GenericStructure.read <lXtractor.core.structure.**
   GenericStructure.read>` to parse the file.
2. **Split into chains using :meth:`split_chains <lXtractor.core.structure.**
   GenericStructure.split_chains>`.
3. **Initialize *ChainStructure* from each chain via**
   `from_structure()`.

```
s = GenericStructure.read(Path("path/to/structure.cif"))
chain_structures = [
    ChainStructure.from_structure(c) for c in s.split_chains()
]
```

Two main containers are:

1) **_seq – a ChainSequence of this structure,**
   also containing meta info.
2) **pdb – a container with pdb id, pdb chain id,**
   and the structure itself.

A unique structure is defined by

__init__(*structure*, *chain_id=None*, *structure_id=None*, *seq=None*, *parent=None*, *children=None*, *variables=None*)

Parameters

- **structure_id** (*str | None*) – An ID for the structure the chain was taken from.
- **chain_id** (*str | None*) – A chain ID (e.g., “A”, “B”, etc.)

- **structure** (GenericStructure | *bst.AtomArray* | *None*) – Parsed generic structure with a single chain.
- **seq** (ChainSequence | *None*) – Chain sequence of a structure. If not provided, will use `get_sequence`.
- **parent** (ChainStructure | *None*) – Specify parental structure.
- **children** (*abc.Iterable[ChainStructure]* | *None*) – Specify structures descended from this one. This contained is used to record sub-structures obtained via *spawn_child()*.
- **variables** (Variables | *None*) – Variables associated with this structure.

Raises
InitError – If invalid (e.g., multi-chain structure) is provided.

apply_children(*fn*, *inplace=False*)

Apply some function to children.

Parameters

- **fn** (ApplyT[ChainStructure]) – A callable accepting and returning the chain structure type instance.
- **inplace** (*bool*) – Apply to children in place. Otherwise, return a copy with only children transformed.

Returns
A chain structure with transformed children.

Return type
t.Self

filter_children(*pred*, *inplace=False*)

Filter children using some predicate.

Parameters

- **pred** (FilterT[ChainStructure]) – Some callable accepting chain structure and returning bool.
- **inplace** (*bool*) – Filter *children* in place. Otherwise, return a copy with only children transformed.

Returns
A chain structure with filtered children.

Return type
t.Self

iter_children()

Iterate *children* in topological order.

See `ChainSequence.iter_children()` and `topo_iter()`.

Return type
Generator[list[ChainStructure], None, None]

classmethod make_empty()

Create an empty chain structure.

Returns
An empty chain structure.

Return type
ChainStructure

classmethod read(*base_dir*, *, *search_children=False*, ***kwargs*)

Read the chain structure from a file disk dump.

Parameters

- **base_dir** (*Path*) – An existing dir containing structure, structure sequence, meta info, and (optionally) any sub-structure segments.
- **dump_names** – File names container.
- **search_children** (*bool*) – Recursively search for sub-segments and populate *children*.
- **kwargs** – Passed to *lXtractor.core.structure.GenericStructure.read()*.

Returns
An initialized chain structure.

Return type
t.Self

rm_solvent(*copy=False*)

Remove solvent "residues" from this structure.

Parameters
copy (*bool*) – Copy an atom array that results from solvent removal.

Returns
A new instance without solvent molecules.

Return type
t.Self

spawn_child(*start*, *end*, *name=None*, *category=None*, *, *map_from=None*, *map_closest=True*, *keep_seq_child=False*, *keep=True*, *deep_copy=False*, *tolerate_failure=False*, *silent=False*)

Create a sub-structure from this one. *Start* and *end* have inclusive boundaries.

Parameters

- **start** (*int*) – Start coordinate.
- **end** (*int*) – End coordinate.
- **name** (*str | None*) – The name of the spawned sub-structure.
- **category** (*str | None*) – Spawned child category. Any meaningful tag string that could be used later to group similar children.
- **map_from** (*str | None*) – Optionally, the map name the boundaries correspond to.
- **map_closest** (*bool*) – Map to closest *start*, *end* boundaries (see `map_boundaries()`).
- **keep_seq_child** (*bool*) – Keep spawned sub-sequence within `ChainSequence.children`. Beware that it's best to use a single object type for keeping parent-children relationships to avoid duplicating information.
- **keep** (*bool*) – Keep spawned substructure in *children*.
- **deep_copy** (*bool*) – Deep copy spawned sub-sequence and sub-structure.
- **tolerate_failure** (*bool*) – Do not raise the ``InitError` if the resulting structure subset is empty,

- **silent** (*bool*) – Do not display warnings if *tolerate_failure* is True.

Returns
New chain structure – a sub-structure of the current one.

Return type
ChainStructure

summary(*meta=True*, *children=False*, *ligands=False*)

Return type
DataFrame

superpose(*other*, *res_id=None*, *atom_names=None*, *map_name_self=None*, *map_name_other=None*, *mask_self=None*, *mask_other=None*, *inplace=False*, *rmsd_to_meta=True*)

Superpose some other structure to this one. It uses func:*biotite.structure.superimpose* internally.

The most important requirement is both structures (after all optional selections applied) having the same number of atoms.

Parameters

- **other** (ChainStructure) – Other chain structure (mobile).
- **res_id** (*Sequence[int] | None*) – Residue positions within this or other chain structure. If None, use all available residues.
- **atom_names** (*Sequence[Sequence[str]] | Sequence[str] | None*) – Atom names to use for selected residues. Two options are available:

  1) Sequence of sequences of atom names. In this case, atom names are given per selected residue (*res_id*), and the external sequence's length must correspond to the number of residues in the *res_id*. Note that if no *res_id* provided, the sequence must encompass all available residues.

  2) A sequence of atom names. In this case, it will be used to select atoms for each available residues. For instance, use `atom_names=["CA", "C", "N"]` to select backbone atoms.
- **map_name_self** (*str | None*) – Use this map to map *res_id* to real numbering of this structure.
- **map_name_other** (*str | None*) – Use this map to map *res_id* to real numbering of the *other* structure.
- **mask_self** (*ndarray | None*) – Per-atom boolean selection mask to pick fixed atoms within this structure.
- **mask_other** (*ndarray | None*) – Per-atom boolean selection mask to pick mobile atoms within the *other* structure. Note that *mask_self* and *mask_other* take precedence over other selection specifications.
- **inplace** (*bool*) – Apply the transformation to the mobile structure inplace, mutating *other*. Otherwise, make a new instance: same as *other*, but with transformed atomic coordinates of a `pdb.structure`.
- **rmsd_to_meta** (*bool*) – Write RMSD to the *meta* of *other* as "rmsd

Returns
A tuple with (1) transformed chain structure, (2) transformation RMSD, and (3) transformation matrices (see func:*biotite.structure.superimpose* for details).

Return type
tuple[ChainStructure, float, tuple[*ndarray*, *ndarray*, *ndarray*]]

write(*dest*, *fmt='mmtf.gz'*, ***, *write_children=False*)

Write this object into a directory. It will create the following files:

1. meta.tsv
2. sequence.tsv
3. structure.fmt

Existing files will be overwritten.

Parameters

- **dest** (*Path*) – A writable dir to save files to.
- **fmt** (*str*) – Structure format to use. Supported formats are "pdb", "cif", and "mmtf". Adding ".gz" (eg, "mmtf.gz") will lead to gzip compression.
- **write_children** (*bool*) – Recursively write *children*.

Returns
Path to the directory where the files are written.

Return type
Path

property altloc: str

Returns
An altloc ID.

property array: AtomArray

Returns
The `AtomArray` object (a shortcut for `.pdb.structure.array`).

property categories: list[str]

Returns
A list of categories encapsulated within `ChainSequence.meta`.

property chain_id: str

children: *ChainList*[*ChainStructure*]

Any sub-structures descended from this one, preferably using *spawn_child()*.

property end: int

Returns
Structure sequence's end

property id: str

Returns
ChainStructure identifier in the format "ChainStructure({_seq.id}|{alt_locs})<-(parent.id)".

property is_empty: bool

Returns
`True` if the structure is empty and `False` otherwise.

property ligands: tuple[*Ligand*, ...]

Returns
A list of connected ligands.

property meta: dict[str, str]

Returns
Meta info of a `_seq`.

property name: str | None

Returns
Structure sequence's `name`

property parent: t.Self | None

property seq: *ChainSequence*

property start: int

Returns
Structure sequence's `start`

property structure: *GenericStructure*

variables: *Variables*

Variables assigned to this structure. Each should be of a *lXtractor.variables.base.StructureVariable*.

lXtractor.chain.structure.**filter_selection_extended**(*c*, *pos=None*, *atom_names=None*, *map_name=None*, *exclude_hydrogen=False*, *tolerate_missing=False*)

Get mask for certain positions and atoms of a chain structure.

Parameters

- **c** (ChainStructure) – Arbitrary chain structure.
- **pos** (*Sequence[int] | None*) – A sequence of positions.
- **atom_names** (*Sequence[Sequence[str]] | Sequence[str] | None*) – A sequence of atom names (broadcasted to each position in *res_id*) or an iterable over such sequences for each position in *res_id*.
- **map_name** (*str | None*) – A map name to map from *pos* to `numbering`
- **exclude_hydrogen** (*bool*) – For convenience, exclude hydrogen atoms. Especially useful during pre-processing for superposition.
- **tolerate_missing** (*bool*) – If certain positions failed to map, does not raise an error.

Returns
A binary mask, `True` for selected atoms.

Return type
ndarray

lXtractor.chain.structure.**subset_to_matching**(*reference*, *c*, *map_name=None*, *skip_if_match='seq1'*, ***kwargs*)

Subset both chain structures to aligned residues using **sequence alignment**.

Note: It's not necessary, but it makes sense for *c1* and *c2* to be somehow related.

Parameters

- **reference** (ChainStructure) – A chain structure to align to.
- **c** (ChainStructure) – A chain structure to align.
- **map_name** (*str | None*) – If provided, *c* is considered "pre-aligned" to the *reference*, and *reference* possessed the numbering under *map_name*.
- **skip_if_match** (*str*) – Two options:
  1. Sequence/Map name, e.g., "seq1" – if sequences under this name match exactly, skip alignment and return original chain structures.
  2. "len" – if sequences have equal length, skip alignment and return original chain structures.

Returns
A pair of new structures having the same number of residues that were successfully matched during the alignment.

Return type
tuple[ChainStructure, ChainStructure]

lXtractor.chain.chain module

class lXtractor.chain.chain.**Chain**(*seq*, *structures=None*, *parent=None*, *children=None*)

Bases: object

A container, encompassing a ChainSequence and possibly many ChainStructure's corresponding to a single protein chain.

A typical use case is when one wants to benefit from the connection of structural and sequential data, e.g., using single full canonical sequence as _seq and all the associated structures within *structures*. In this case, this data structure makes it easier to extract, annotate, and calculate variables using canonical sequence mapped to the sequence of a structure.

Typical workflow:

1. Initialize from some canonical sequence.
2. Add structures and map their sequences.
3. ???
4. **Do something useful, like calculate variables using canonical** sequence's positions.

```
c = Chain.from_sequence((header, _seq))
for s in structures:
    c.add_structure(s)
```

__init__(*seq*, *structures=None*, *parent=None*, *children=None*)

Parameters

- **seq** (ChainSequence) – A chain sequence.
- **structures** (*Iterable[ChainStructure] | None*) – Chain structures corresponding to a single protein chain specified by *_seq*.
- **parent** (Chain | *None*) – A parent chain this chain had descended from.
- **children** (*Iterable[Chain] | None*) – A collection of children.

add_structure(*structure*, ***, *check_ids=True*, *map_to_seq=True*, *map_name='map_canonical'*, *add_to_children=False*, ***kwargs*)

Add a structure to *structures*.

Parameters

- **structure** (ChainStructure) – A structure of a single chain corresponding to `_seq`.
- **check_ids** (*bool*) – Check that existing *structures* don't encompass the structure with the same *id()*.
- **map_to_seq** (*bool*) – Align the structure sequence to the `_seq` and create a mapping within the former.
- **map_name** (*str*) – If *map_to_seq* is `True`, use this map name.
- **add_to_children** (*bool*) – If `True`, will recursively add structure to existing children according to their boundaries mapped to the structure's numbering. Consequently, this requires mapping, i.e., `map_to_seq=True`.
- **kwargs** – Passed to `ChainSequence.map_numbering()`.

Returns
Mutates *structures* and returns nothing.

Raises
ValueError – If *check_ids* is `True` and the structure id clashes with the existing ones.

apply_children(*fn*, *inplace=False*)

Apply some function to children.

Parameters

- **fn** (ApplyT[Chain]) – A callable accepting and returning the chain type instance.
- **inplace** (*bool*) – Apply to children in place. Otherwise, return a copy with only children transformed.

Returns
A chain with transformed children.

Return type
t.Self

apply_structures(*fn*, *inplace=False*)

Apply some function to *structures*.

Parameters

- **fn** (ApplyT[ChainStructure]) – A callable accepting and returning a chain structure.
- **inplace** (*bool*) – Apply to *structures* in place. Otherwise, return a copy with only children transformed.

Returns
A chain with transformed structures.

Return type
t.Self

filter_children(*pred*, *inplace=False*)

Filter children using some predicate.

Parameters

- **pred** (*FilterT[Chain]*) – Some callable accepting chain and returning bool.
- **inplace** (*bool*) – Filter *children* in place. Otherwise, return a copy with only children transformed.

Returns
A chain with filtered children.

Return type
t.Self

filter_structures(*pred*, *inplace=False*)

Filter chain *structures*.

Parameters

- **pred** (*FilterT[ChainStructure]*) – A callable accepting a chain structure and returning bool.
- **inplace** (*bool*) – Filter *structures* in place. Otherwise, return a copy with only children transformed.

Returns
A chain with filtered structures.

Return type
t.Self

generate_patched_seqs(*numbering='numbering'*, *link_name='map_canonical'*, *link_points_to='i'*, ***kwargs*)

Generate patched sequences from chain structure sequences.

For explanation of the patching process see *lXtractor.chain.sequence.ChainSequence.patch()*.

Parameters

- **numbering** (*str*) – Map name referring to a numbering scheme to infer gaps from.
- **link_name** (*str*) – Map name linking structure sequence to the canonical sequence.
- **link_points_to** (*str*) – Map name in the canonical sequence that *link_name* refers to.
- **kwargs** – Passed to *lXtractor.chain.sequence.ChainSequence.patch()*.

Returns
A generator over patched structure sequences.

Return type
Generator[ChainSequence, None, None]

iter_children()

Iterate *children* in topological order.

See `ChainSequence.iter_children()` and `topo_iter()`.

Returns
Iterator over levels of a child tree.

Return type
Generator[list[Chain], None, None]

classmethod make_empty()

Return type
t.Self

classmethod read(*path*, *, *search_children=False*)

Parameters

- **path** (*Path*) – A path to a directory with at least sequence and metadata files.
- **search_children** (*bool*) – Recursively search for child segments and populate *children*.

Returns
An initialized chain.

Return type
Chain

spawn_child(*start*, *end*, *name=None*, *category=None*, *, *subset_structures=True*, *tolerate_failure=False*, *silent=False*, *keep=True*, *seq_deep_copy=False*, *seq_map_from=None*, *seq_map_closest=True*, *seq_keep_child=False*, *str_deep_copy=False*, *str_map_from=None*, *str_map_closest=True*, *str_keep_child=True*, *str_seq_keep_child=False*, *str_min_size=1*, *str_accept_fn=<function Chain.<lambda>>*)

Subset a `_seq` and (optionally) each structure in *structures* using the provided `_seq` boundaries (inclusive).

Parameters

- **start** (*int*) – Start coordinate.
- **end** (*int*) – End coordinate.
- **name** (*str | None*) – Name of a new chain.
- **category** (*str | None*) – Spawned child category. Any meaningful tag string that could be used later to group similar children.
- **subset_structures** (*bool*) – If `True`, subset each structure in *structures*. If `False`, structures are not inherited.
- **tolerate_failure** (*bool*) – If `True`, a failure to subset a structure doesn't raise an error.
- **silent** (*bool*) – Supress warnings for errors when *tolerate_failure* is `True`.
- **keep** (*bool*) – Save created child to *children*.
- **seq_deep_copy** (*bool*) – Deep copy potentially mutable sequences within `_seq`.
- **seq_map_from** (*str | None*) – Use this map to obtain coordinates within `_seq`.
- **seq_map_closest** (*bool*) – Map to the closest matching coordinates of a `_seq`. See `ChainSequence.map_boundaries()` and `ChainSequence.find_closest()`.
- **seq_keep_child** (*bool*) – Keep a spawned `ChainSequence` as a child within `_seq`. Should be `False` if *keep* is `True` to avoid data duplication.
- **str_deep_copy** (*bool*) – Deep copy each sub-structure.
- **str_map_from** (*str | None*) – Use this map to obtain coordinates within `ChainStructure._seq` of each structure.

- **str_map_closest** (*bool*) – Map to the closest matching coordinates of a `_seq`. See `ChainSequence.map_boundaries()` and `ChainSequence.find_closest()`.
- **str_keep_child** (*bool*) – Keep a spawned sub-structure as a child in `ChainStructure.children`. Should be `False` if *keep* is `True` to avoid data duplication.
- **str_seq_keep_child** (*bool*) – Keep a sub-sequence of a spawned structure within the `ChainSequence.children` of `ChainStructure._seq` of a spawned structure. Should be `False` if *keep* or *str_keep_child* is `True` to avoid data duplication.
- **str_min_size** (*int | float*) – A minimum number of residues in a structure to be accepted after subsetting.
- **str_accept_fn** (*abc.Callable[[ChainStructure], bool]*) – A filter function accepting a *ChainStructure* and returning a boolean value indicating whether this structure should be retained in *structures*.

Returns
A sub-chain with sub-sequence and (optionally) sub-structures.

Return type
t.Self

summary(*meta=True*, *children=False*, *structures=True*)

Return type
DataFrame

transfer_seq_mapping(*map_name*, *link_map='map_canonical'*, *link_map_points_to='i'*, ***kwargs*)

Transfer sequence mapping to each `ChainStructure._seq` within *structures*.

This method simply utilizes `ChainSequence.relate()` to transfer some map from the `_seq` to each `ChainStructure._seq`. Check `ChainSequence.relate()` for an explanation.

Parameters

- **map_name** (*str*) – The name of the map to transfer.
- **link_map** (*str*) – A name of the map existing within `ChainStructure._seq` of each structure in *structures*.
- **link_map_points_to** (*str*) – Which sequence values of the *link_map* point to.
- **kwargs** – Passed to `ChainSequence.relate()`

Returns
Nothing.

write(*dest*, ***, *str_fmt='mmtf.gz'*, *write_children=True*)

Create a disk dump of this chain data. Created dumps can be reinitialized via *read()*.

Parameters

- **dest** (*Path*) – A writable dir to hold the data.
- **str_fmt** (*str*) – A format to write *structures* in.
- **write_children** (*bool*) – Recursively write *children*.

Returns
Path to the directory where the files are written.

Return type
Path

property categories: list[str]

Returns
A list of categories from `_seq`'s `ChainSequence.meta`.

children: *ChainList*[*Chain*]

A collection of children preferably obtained using *spawn_child()*.

property end: int

Returns
Structure sequence's `end`

property id: str

Returns
Chain identifier derived from its `_seq` ID.

property meta: dict[str, str]

Returns
A *seq()*'s `ChainSequence.meta`.

property name: str | None

Returns
Structure sequence's `name`

property parent: t.Self | None

property seq: *ChainSequence*

property start: int

Returns
Structure sequence's `start`

structures: *ChainList*[*ChainStructure*]

lXtractor.chain.list module

The module defines the *ChainList* - a list of *Chain**-type objects that behaves like a regular list but has additional bells and whistles tailored towards *Chain** data structures.

class `lXtractor.chain.list.`**ChainList**(*chains*, *categories=None*)

Bases: `MutableSequence[CT]`

A mutable single-type collection holding either `Chain`'s, or `ChainSequence`'s, or `ChainStructure`'s.

Object's funtionality relies on this type purity. Adding of / contatenating with objects of a different type shall raise an error.

It behaves like a regular list with additional functionality.

```
>>> from lXtractor.chain import ChainSequence
>>> s = ChainSequence.from_string('SEQUENCE', name='S')
>>> x = ChainSequence.from_string('XXX', name='X')
>>> x.meta['category'] = 'x'
>>> cl = ChainList([s, s, x])
```

(continues on next page)

(continued from previous page)

```
>>> cl
[S|1-8, S|1-8, X|1-3]
>>> cl[0]
S|1-8
>>> cl['S']
[S|1-8, S|1-8]
>>> cl[:2]
[S|1-8, S|1-8]
>>> cl['1-3']
[X|1-3]
```

Adding/appending/removing objects of a similar type is easy and works similar to a regular list.

```
>>> cl += [s]
>>> assert len(cl) == 4
>>> cl.remove(s)
>>> assert len(cl) == 3
```

Categories can be accessed as attributes or using `[]` syntax (similar to the *Pandas.DataFrame* columns).

```
>>> cl.x
[X|1-3]
>>> cl['x']
[X|1-3]
```

While creating a chain list, using a *groups* parameter will assign categories to sequences. Note that such operations return a new *ChainList* object.

```
>>> cl = ChainList([s, x], categories=['S', ['X1', 'X2']])
>>> cl.S
[S|1-8]
>>> cl.X2
[X|1-3]
>>> cl['X1']
[X|1-3]
```

__init__(*chains*, *categories=None*)

Parameters

- **chains** (*Iterable[CT]*) – An iterable over Chain*-type objects.
- **categories** (*Iterable[str | Iterable[str]] | None*) – An optional list of categories. If provided, they will be assigned to inputs' *meta* attributes.

apply(*fn*, *verbose=False*, *desc='Applying to objects'*, *num_proc=1*)

Apply a function to each object and return a new chain list of results.

Parameters

- **fn** (ApplyT) – A callable to apply.
- **verbose** (*bool*) – Display progress bar.

- **desc** (*str*) – Progress bar description.
- **num_proc** (*int*) – The number of CPUs to use. `num_proc <= 1` indicates sequential processing.

Returns
A new chain list with application results.

Return type
ChainList[*CT*]

collapse()

Collapse all objects and their children within this list into a new chain list. This is a shortcut for `chain_list + chain_list.collapse_children()`.

Returns
Collapsed list.

Return type
ChainList[*CT*]

collapse_children()

Collapse all children of each object in this list into a single chain list.

```
>>> from lXtractor.chain import ChainSequence
>>> s = ChainSequence.from_string('ABCDE', name='A')
>>> child1 = s.spawn_child(1, 4)
>>> child2 = child1.spawn_child(2, 3)
>>> cl = ChainList([s]).collapse_children()
>>> assert isinstance(cl, ChainList)
>>> cl
[A|1-4<-(A|1-5), A|2-3<-(A|1-4<-(A|1-5))]
```

Returns
A chain list of all children.

Return type
ChainList[*CT*]

drop_duplicates(*key=<function ChainList.<lambda>>*)

Parameters
key (*abc.Callable[[CT], t.Hashable] | None*) – A callable accepting the single element and returning some hashable object associated with that element.

Returns
A new list with unique elements as judged by the *key*.

Return type
t.Self

filter(*pred*)

```
>>> from lXtractor.chain import ChainSequence
>>> cl = ChainList(
...     [ChainSequence.from_string('AAAX', name='A'),
...      ChainSequence.from_string('XXX', name='X')]
... )
```

(continues on next page)

(continued from previous page)

```
>>> cl.filter(lambda c: c.seq1[0] == 'A')
[A|1-4]
```

Parameters
pred (*Callable[[CT], bool]*) – Predicate callable for filtering.

Returns
A filtered chain list (new object).

Return type
ChainList[*CT*]

filter_category(*name*)

Parameters
name (*str*) – Category name.

Returns
Filtered objects having this category within their `meta["category"]`.

Return type
ChainList

filter_pos(*s*, *, *match_type='overlap'*, *map_name=None*)

Filter to objects encompassing certain consecutive position regions or arbitrary positions' collections.

For `Chain` and `ChainStructure`, the filtering is over *_seq* attributes.

Parameters

- **s** (*lxs.Segment | abc.Collection[Ord]*) – What to search for:
  1. **s=Segment(start, end) to find all objects encompassing** certain region.
  2. **[pos1, posX, posN] to find all objects encompassing the** specified positions.
- **match_type** (*str*) – If *s* is *Segment*, this value determines the acceptable relationships between *s* and each `ChainSequence`:
  1. "overlap" – it's enough to overlap with *s*.
  2. "bounding" – object is accepted if it bounds *s*.
  3. "bounded" – object is accepted if it's bounded by *s*.
- **map_name** (*str | None*) – Use this map within to map positions of *s*. For instance, to each for all elements encompassing region 1-5 of a canonical sequence, one would use

```
chain_list.filter_pos(
    s=Segment(1, 5), match_type="bounding",
    map_name="map_canonical"
)
```

Returns
A list of hits of the same type.

Return type
ChainList[CS]

get_level(*n*)

Get a specific level of a hierarchical tree starting from this list:

```
l0: this list
l1: children of each child of each object in l0
l2: children of each child of each object in l1
...
```

Parameters
n (*int*) – The level index (0 indicates this list). Other levels are obtained via *iter_children()*.

Returns
A chain list of object corresponding to a specific topological level of a child tree.

Return type
ChainList[*CT*]

groupby(*key*)

Group sequences in this list by a given key.

Parameters
key (*abc.Callable[[CT], T]*) – Some callable accepting a single chain and returning a grouper value.

Returns
An iterator over pairs `(group, chains)`, where `chains` is a chain list of chains that belong to `group`.

Return type
abc.Iterator[tuple[T, t.Self]]

index(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

Return type
int

insert(*index*, *value*)

S.insert(index, value) – insert value before index

iter_children()

Simultaneously iterate over topological levels of children.

```
>>> from lXtractor.chain import ChainSequence
>>> s = ChainSequence.from_string('ABCDE', name='A')
>>> child1 = s.spawn_child(1, 4)
>>> child2 = child1.spawn_child(2, 3)
>>> x = ChainSequence.from_string('XXXX', name='X')
>>> child3 = x.spawn_child(1, 3)
>>> cl = ChainList([s, x])
>>> list(cl.iter_children())
[[A|1-4<-(A|1-5), X|1-3<-(X|1-4)], [A|2-3<-(A|1-4<-(A|1-5))]]
```

Returns
An iterator over chain lists of children levels.

Return type
Generator[ChainList[*CT*], None, None]

`iter_ids()`

Iterate over ids of this chain list.

Returns
An iterator over chain ids.

Return type
Iterator[str]

`iter_sequences()`

Returns
An iterator over `ChainSequence`'s.

Return type
abc.Generator[*ChainSequence*, None, None]

`iter_structure_sequences()`

Returns
Iterate over `ChainStructure._seq` attributes.

Return type
abc.Generator[*ChainSequence*, None, None]

`iter_structures()`

Returns
An generator over `ChainStructure`'s.

Return type
abc.Generator[*ChainStructure*, None, None]

`sort(key=<function ChainList.<lambda>>)`

Return type
ChainList[*CT*]

`summary(**kwargs)`

Return type
DataFrame

`property categories: Set[str]`

Returns
A set of categories inferred from *meta* of encompassed objects.

`property ids: list[str]`

Returns
A list of ids for all chains in this list.

property sequences: *ChainList*[*ChainSequence*]

Returns
Get all `lXtractor.core.chain.Chain._seq` or *lXtractor.core.chain.sequence.ChainSequence* objects within this chain list.

property structure_sequences: *ChainList*[*ChainSequence*]

property structures: *ChainList*[*ChainStructure*]

lXtractor.chain.list.**add_category**(*c*, *cat*)

Parameters

- **c** (*Any*) – A Chain*-type object.
- **cat** (*str*) – Category name.

Returns

lXtractor.chain.io module

class lXtractor.chain.io.**ChainIO**(*num_proc=1*, *verbose=False*, *tolerate_failures=False*)

Bases: `object`

A class handling reading/writing collections of *Chain** objects.

__init__(*num_proc=1*, *verbose=False*, *tolerate_failures=False*)

Parameters

- **num_proc** (*int*) – The number of parallel processes. Using more processes is especially beneficial for `ChainStructure`'s and `Chain`'s with structures. Otherwise, the increasing this number may not reduce or actually worsen the time needed to read/write objects.
- **verbose** (*bool*) – Output logging and progress bar.
- **tolerate_failures** (*bool*) – Errors when reading/writing do not raise an exception.

read(*obj_type*, *path*, *callbacks=()*, ***kwargs*)

Read `obj_type`-type objects from a path or an iterable of paths.

Parameters

- **obj_type** (*Type[CT]*) – Some class with `@classmethod(read(path))`.
- **path** (*Path | Iterable[Path]*) – Path to the dump to read from. It's a path to directory holding files necessary to init a given *obj_type*, or an iterable over such paths.
- **callbacks** (*Sequence[Callable[[CT], CT]]*) – Callables applied sequentially to parsed object.
- **kwargs** – Passed to the object's *read()* method.

Returns
A generator over initialized objects or futures.

Return type
Generator[*CT* | None, None, None]

read_chain(*path*, ***kwargs*)

Read `Chain`'s from the provided path.

If *path* contains signature files and directories (such as *sequence.tsv* and *segments*), it is assumed to contain a single object. Otherwise, it is assumed to contain multiple `Chain` objects.

Parameters

- **path** (*Path | Iterable[Path]*) – Path to a dump or a dir of dumps.
- **kwargs** – Passed to *read()*.

Returns
An iterator over `Chain` objects.

Return type
Generator[Chain | None, None, None]

read_chain_seq(*path*, ***kwargs*)

Read `ChainSequence`'s from the provided path.

If *path* contains signature files and directories (such as *sequence.tsv* and *segments*), it is assumed to contain a single object. Otherwise, it is assumed to contain multiple `ChainSequence` objects.

Parameters

- **path** (*Path | Iterable[Path]*) – Path to a dump or a dir of dumps.
- **kwargs** – Passed to *read()*.

Returns
An iterator over `ChainSequence` objects.

Return type
Generator[ChainSequence | None, None, None]

read_chain_str(*path*, ***kwargs*)

Read `ChainStructure`'s from the provided path.

If *path* contains signature files and directories (such as *structure.cif* and *segments*), it is assumed to contain a single object. Otherwise, it is assumed to contain multiple `ChainStructure` objects.

Parameters

- **path** (*Path | Iterable[Path]*) – Path to a dump or a dir of dumps.
- **kwargs** – Passed to *read()*.

Returns
An iterator over `ChainStructure` objects.

Return type
Generator[ChainStructure | None, None, None]

write(*chains*, *base*, *overwrite=False*, ***kwargs*)

Parameters

- **chains** (*CT | Iterable[CT]*) – A single or multiple chains to write.
- **base** (*Path*) – A writable dir. For multiple chains, will use *base/chain.id* directory.
- **overwrite** (*bool*) – If the destination folder exists, `False` means returning the destination path without attempting to write the chain, whereas `True` results in an explicit `.write()` call.

- **kwargs** – Passed to a chain's *write* method.

Returns
Whatever *write* method returns.

Return type
Generator[*Path* | None | *Future*, None, None]

num_proc

The number of parallel processes

tolerate_failures

Errors when reading/writing do not raise an exception.

verbose

Output logging and progress bar.

class lXtractor.chain.io.**ChainIOConfig**(*num_proc: 'int' = 1*, *verbose: 'bool' = False*, *tolerate_failures: 'bool' = False*)

Bases: object

__init__(*num_proc=1*, *verbose=False*, *tolerate_failures=False*)

num_proc: int = 1

tolerate_failures: bool = False

verbose: bool = False

lXtractor.chain.io.**read_chains**(*paths*, *children*, ***, *seq_cfg=ChainIOConfig(num_proc=1, verbose=False, tolerate_failures=False)*, *str_cfg=ChainIOConfig(num_proc=1, verbose=False, tolerate_failures=False)*, *seq_callbacks=()*, *str_callbacks=()*, *seq_kwargs=None*, *str_kwargs=None*)

Reads saved `lXtractor.core.chain.chain.Chain` objects without invoking `lXtractor.core.chain.chain.Chain.read()`. Instead, it will use separate *ChainIO* instances to read chain sequences and chain structures. The output is identical to *ChainIO.read_chain_seq()*.

Consider using it for:

1. For parallel parsing of `Chain` objects with many structures.
2. For separate treatment of chain sequences and chain structures.
3. For better customization of chain sequences and structures parsing.

Parameters

- **paths** (*Path | Sequence[Path]*) – A path or a sequence of paths to chains.
- **children** (*bool*) – Search for, parse and integrate all nested children.
- **seq_cfg** (ChainIOConfig) – *ChainIO* config for chain sequences parsing.
- **str_cfg** (ChainIOConfig) – ... for chain structures parsing.
- **seq_callbacks** (*Sequence[Callable[[CT], CT]]*) – A (potentially empty) sequence passed to the reader. Each callback must accept and return a single chain sequence.
- **str_callbacks** (*Sequence[Callable[[CT], CT]]*) – ... Same for the structures.

- **seq_kwargs** (*dict[str, Any] | None*) – Passed to `lXtractor.core.chain.sequence.ChainSequence.read()`.
- **str_kwargs** (*dict[str, Any] | None*) – Passed to `lXtractor.core.chain.structure.ChainStructure.read()`.

Returns
A chain list of parsed chains.

Return type
ChainList[Chain]

lXtractor.chain.initializer module

A module encompassing the *ChainInitializer* used to init `Chain*`-type objects from various input types. It enables parallelization of reading structures and seq2seq mappings and is flexible thanks to callbacks.

class lXtractor.chain.initializer.**ChainInitializer**(*tolerate_failures=False*, *verbose=False*)

Bases: `object`

In contrast to `ChainIO`, this object initializes new `Chain`, `ChainStructure`, or `Chain` objects from various input types.

To initialize `Chain` objects, use *from_mapping()*.

To initialize `ChainSequence` or `ChainStructure` objects, use *from_iterable()*.

__init__(*tolerate_failures=False*, *verbose=False*)

Parameters

- **tolerate_failures** (*bool*) – Don't stop the execution if some object fails to initialize.
- **verbose** (*bool*) – Output progress bars.

from_iterable(*it*, *num_proc=1*, *callbacks=None*, *desc='Initializing objects'*)

Initialize `ChainSequence`s or/and :class:`ChainStructure`'s from (possibly heterogeneous) iterable.

Parameters

- **it** (*abc.Iterable[ChainSequence | ChainStructure | Path | tuple[Path, abc.Sequence[str]] | tuple[str, str] | GenericStructure]*) –

  Supported elements are:

  1) Initialized objects (passed without any actions).
  2) Path to a sequence or a structure file.
  3) (Path to a structure file, list of target chains).
  4) A pair (header, _seq) to initialize a `ChainSequence`.
  5) A `GenericStructure` with a single chain.

- **num_proc** (*int*) – The number of processes to use.
- **callbacks** (*abc.Sequence[SingletonCallback] | None*) – A sequence of callables accepting and returning an initialized object.

- **desc** (*str*) – Progress bar description used if `verbose` is `True`.

Returns
A generator yielding initialized chain sequences and structures parsed from the inputs.

Return type
abc.Generator[_O | Future, None, None]

from_mapping(*m*, *key_callbacks=None*, *val_callbacks=None*, *item_callbacks=None*, ***, *map_numberings=True*, *num_proc_read_seq=1*, *num_proc_read_str=1*, *num_proc_item_callbacks=1*, *num_proc_map_numbering=1*, *num_proc_add_structure=1*, ***kwargs*)

Initialize `Chain`'s from mapping between sequences and structures.

It will first initialize objects to which the elements of *m* refer (see below) and then create maps between each sequence and associated structures, saving these into structure `ChainStructure._seq`'s.

Note: `key/value_callback` are distributed to parser and applied right after parsing the object. As a result, their application will be parallelized depending on the``num_proc_read_seq`` and `num_proc_read_str` parameters.

Parameters

- **m** (*abc.Mapping[ChainSequence | Chain | tuple[str, str] | Path, abc.Sequence[ChainStructure | GenericStructure | bst.AtomArray | Path | tuple[Path, abc.Sequence[str]]]]*) – A mapping of the form {*_seq* => [structures]}, where *_seq* is one of:

  1) Initialized `ChainSequence`.

  2) A pair (header, _seq).

  3) A path to a **fasta** file containing a single sequence.

  While each structure is one of:

  1) Initialized `ChainStructure`.

  2) `GenericStructure` with a single chain.

  3) `biotite.AtomArray` corresponding to a single chain.

  4) A path to a structure file.

  5) (A path to a structure file, list of target chains).

  In the latter two cases, the chains will be expanded and associated with the same sequence.

- **key_callbacks** (*abc.Sequence[SingletonCallback] | None*) – A sequence of callables accepting and returning a `ChainSequence`.

- **val_callbacks** (*abc.Sequence[SingletonCallback] | None*) – A sequence of callables accepting and returning a `ChainStructure`.

- **item_callbacks** (*abc.Sequence[ItemCallback] | None*) – A sequence of callables accepting and returning a parsed item – a tuple of `Chain` and a sequence of associated `ChainStructure`s. Callbacks are applied sequentially to each item as a function composition in the supplied order (left to right). It the last callback returns ``None` as a first element or an empty list as a second element, such item will be filtered out. Item callbacks are

applied after parsing sequences and structures and converting chain sequences to chains.

- **map_numberings** (*bool*) – Map PDB numberings to canonical sequence's numbering via pairwise sequence alignments.
- **num_proc_read_seq** (*int*) – A number of processes to devote to sequence parsing. Typically, sequence reading doesn't benefit from parallel processing, so it's better to leave this default.
- **num_proc_read_str** (*int*) – A number of processes dedicated to structures parsing.
- **num_proc_item_callbacks** (*int*) – A number of CPUs to parallelize item callbacks' application.
- **num_proc_map_numbering** (*int*) – A number of processes to use for mapping between numbering of sequences and structures. Generally, this should be as high as possible for faster processing. In contrast to the other operations here, this one seems more CPU-bound and less resource hungry (although, keep in mind the size of the canonical sequence: if it's too high, the RAM usage will likely explode). If `None`, will default to `num_proc`.
- **num_proc_add_structure** (*int*) – In case of parallel numberings mapping, i.e, when `num_proc_map_numbering > 1`, this option allows to transfer these numberings and add structures to chains in parallel. It may be useful to when `add_to_children=True` is passed in `kwargs` as it allows creating sub-structures in parallel.
- **kwargs** – Passed to `Chain.add_structure()`.

Returns
A list of initialized chains.

Return type
ChainList[*Chain*]

property supported_seq_ext: list[str]

Returns
Supported sequence file extensions.

property supported_str_ext: list[str]

Returns
Supported structure file extensions.

class lXtractor.chain.initializer.**ItemCallback**(**args*, ***kwargs*)

Bases: `Protocol`

A callback applied to processed items in *ChainInitializer.from_mapping()*.

__call__(*inp*)

Call self as a function.

Return type
tuple[*Chain* | None, list[*ChainStructure*]]

__init__(**args*, ***kwargs*)

class lXtractor.chain.initializer.**SingletonCallback**(**args*, ***kwargs*)

Bases: Protocol

A protocol defining signature for a callback used with *ChainInitializer* on single objects right after parsing.

__call__(*inp: CT*) → CT | None

__call__(*inp: list[*ChainStructure*]*) → list[*ChainStructure*] | None

__call__(*inp: None*) → None

Call self as a function.

__init__(**args*, ***kwargs*)

lXtractor.chain.tree module

A module to handle the ancestral tree of the Chain*-type objects defined by their `parent`/`children` attributes and/or `meta` info.

lXtractor.chain.tree.**list_ancestors**(*c*)

```
>>> o = ChainSequence.from_string('x' * 5, 1, 5, 'C')
>>> c13 = o.spawn_child(1, 3)
>>> c12 = c13.spawn_child(1, 2)
>>> list_ancestors(c12)
[C|1-3<-(C|1-5), C|1-5]
```

Parameters
c (Chain | ChainSequence | ChainStructure) – Chain*-type object.

Returns
A list ancestor objects obtained from the `parent` attribute..

Return type
list[Chain | ChainSequence | ChainStructure]

lXtractor.chain.tree.**list_ancestors_names**(*id_or_chain*)

```
>>> list_ancestors_names('C|1-5<-(C|1-3<-(C|1-2))')
['C|1-3', 'C|1-2']
```

Parameters
id_or_chain (Chain | ChainSequence | ChainStructure | *str*) – Chain*-type object or its id.

Returns
A list of parents '{name}|{start}-{end}' representations parsed from the object's id.

Return type
list[str]

lXtractor.chain.tree.**make**(*chains*, *connect=False*, *objects=False*, *check_is_tree=True*)

Make an ancestral tree – a directed graph representing ancestral relationships between chains.

Parameters

- **chains** (*Iterable[Chain | ChainSequence | ChainStructure]*) – An iterable of Chain*-type objects.
- **connect** (*bool*) – Connect actual objects by populating `.children` and `.parent` attributes.
- **objects** (*bool*) – Create an object tree using *make_obj_tree()*. Otherwise, create a "string" tree using *make_str_tree()*. Check the docs of these functions to understand the differences.
- **check_is_tree** (*bool*) – If `True`, check if the obtained graph is actually a tree. If it's not, raise `ValueError`.

Returns

Return type
DiGraph

lXtractor.chain.tree.**make_filled**(*name*, *_t*)

Make a "filled" version of an object to occupy the tree.

Parameters

- **name** (*str*) – Name of the node obtained via `node_name()`.
- **_t** (*CT | Type[CT]*) – Some Chain*-type object.

Returns
An object with filled sequence. If it's a `ChainStructure` object, it will have an empty structure.

Return type
CT

lXtractor.chain.tree.**make_obj_tree**(*chains*, *connect=False*, *check_is_tree=True*)

Make an ancestral tree – a directed graph representing ancestral relationships between chains. The nodes of the tree are Chain*-type objects. Hence, they must be hashable. This restricts types of sequences valid for `ChainSequence` to `abc.Sequence[abc.Hashable]`.

As a useful side effect, this function can aid in filling the gaps in the actual tree indicated by the id-relationship suggested by the "id" field of the `meta` property. In other words, if a segment S|1-2 was obtained by spawning from S|1-5, S|1-2's id will reflect this:

```
>>> s = make_filled('S|1-5', ChainSequence.make_empty())
>>> c12 = s.spawn_child(1, 2)
>>> c12
S|1-2<-(S|1-5)
```

However, if S|1-5 was lost (e.g., by writing/reading S|1-2 to/from disk), and S|1-2.parent is None, we can use ID stored in meta to recover ancestral relationships. This function will attend to such cases and create a filler object S|1-5 with a "*"-filled sequence.

```
>>> c12.parent = None
>>> c12
S|1-2
>>> c12.meta['id']
'S|1-2<-(S|1-5)'
>>> ct = make_obj_tree([c12],connect=True)
>>> assert len(ct.nodes) == 2
```

(continues on next page)

(continued from previous page)

```
>>> [n.id for n in ct.nodes]
['S|1-2<-(S|1-5)', 'S|1-5']
```

Parameters

- **chains** (*Iterable[CT]*) – A homogeneous iterable of Chain*-type objects.
- **connect** (*bool*) – If `True`, connect both supplied and created filler objects via `children` and `parent` attributes.
- **check_is_tree** (*bool*) – If `True`, check if the obtained graph is actually a tree. If it's not, raise `ValueError`.

Returns
A networkx's directed graph with Chain*-type objects as nodes.

Return type
DiGraph

lXtractor.chain.tree.**make_str_tree**(*chains*, *connect=False*, *check_is_tree=True*)

A computationally cheaper alternative to *make_obj_tree()*, where nodes are string objects, while actual objects reside in a node attribute "objs". It allows for a faster tree construction since it avoids expensive hashing of Chain*-type objects.

Parameters

- **chains** (*Iterable[Chain | ChainSequence | ChainStructure]*) – An iterable of Chain*-type objects.
- **connect** (*bool*) – If `True`, connect both supplied and created filler objects via `children` and `parent` attributes.
- **check_is_tree** (*bool*) – If `True`, check if the obtained graph is actually a tree. If it's not, raise `ValueError`.

Returns
A networkx's directed graph.

Return type
DiGraph

lXtractor.chain.tree.**recover**(*c*)

Recover ancestral relationships of a Chain*-type object. This will use *make_str_tree()* to recover ancestors from object IDs of an object itself and any encompassed children.

..note ::
It may be used as a callback in *lXtractor.chain.io.ChainIO.read()*

..note ::
make_str_tree() creates "filled" parents via *make_filled()*

Parameters
c (Chain | ChainSequence | ChainStructure) – A Chain*-type object.

Returns
The same object with populated `children` and `parent` attributes.

Return type
Chain | ChainSequence | ChainStructure

3.1.3 lXtractor.ext package

lXtractor.ext.base module

Base utilities for the ext module, e.g., base classes and common functions.

class lXtractor.ext.base.**ApiBase**(*url_getters*, *max_trials=1*, *num_threads=None*, *verbose=False*)

Bases: object

Base class for simple APIs for webservices.

__init__(*url_getters*, *max_trials=1*, *num_threads=None*, *verbose=False*)

Parameters

- **url_getters** (*dict[str, UrlGetter]*) – A dictionary holding functions constructing urls from provided args.
- **max_trials** (*int*) – Max number of fetching attempts for a given query (PDB ID).
- **num_threads** (*int | None*) – The number of threads to use for parallel requests. If None,will send requests sequentially.
- **verbose** (*bool*) – Display progress bar.

max_trials: int

Upper limit on the number of fetching attempts.

num_threads: int | None

The number of threads passed to the ThreadPoolExecutor.

property url_args: list[tuple[str, list[str]]]

Returns

A list of services and argument names necessary to construct a valid url.

url_getters: dict[str, *UrlGetter*]

A dictionary holding functions constructing urls from provided args.

property url_names: list[str]

Returns

A list of supported services.

verbose: bool

Display progress bar.

class lXtractor.ext.base.**StructureApiBase**(*url_getters*, *max_trials=1*, *num_threads=None*, *verbose=False*)

Bases: *ApiBase*

A generic abstract API to fetch structures and associated info.

Child classes must implement *supported_str_formats()* and have a url constructor named “structures” in url_getters.

fetch_info(*service_name*, *url_args*, *dir_*, ***, *overwrite=False*, *callback=<function load_json_callback>*)

Fetch text information.

Parameters

- **service_name** (*str*) – The name of the service to get a *url_getter* from `url_getters`.
- **dir** – Dir to save files to. If `None`, will keep downloaded files as strings.
- **url_args** (*Iterable[_ArgT]*) – Arguments to a *url_getter*.
- **overwrite** (*bool*) – Overwrite existing files if *dir_* is provided.
- **callback** (*Callable[[_ArgT, _RT], _T] | None*) – Callback to apply after fetching the information file. By default, the content is assumed to be in `json` format. Thus, the default callback will parse the fetched content as `dict`. To disable this behavior, pass `callback=None`.

Returns

A tuple with fetched and remaining inputs. Fetched inputs are tuples, where the first element is the original arguments and the second argument is the dictionary with downloaded data. Remaining inputs are arguments that failed to fetch.

Return type

tuple[list[tuple[*_ArgT*, dict | *Path*]], list[*_ArgT*]]

fetch_structures(*ids*, *dir_*, *fmt='cif'*, ***, *overwrite=False*, *parse=False*, *callback=None*)

Fetch structure files.

PDB example:

See also:

`lXtractor.util.io.fetch_files()`.

Hint: Callbacks will apply in parallel if `num_threads` is above 1.

Note: If the provided callback fails, it is equivalent to the fetching failure and will be presented as such. Initializing in verbose mode will output the stacktrace.

Reading structures and parsing immediately requires using `callback`. Such callback may be partially evaluated *lXtractor.core.structure.GenericStructure.read()* encapsulating the correct format.

Parameters

- **ids** (*Iterable[str]*) – An iterable over structure IDs.
- **dir** – Dir to save files to. If `None`, will keep downloaded files as strings.
- **fmt** (*str*) – Structure format. See *supported_str_formats()*. Adding *.gz* will fetch gzipped files.
- **overwrite** (*bool*) – Overwrite existing files if *dir_* is provided.
- **parse** (*bool*) – If `dir_ is None`, use `parse_callback(fmt=fmt)()` to parse fetched structures right away. This will override any existing *callback*.
- **callback** (*Callable[[tuple[str, str], _RT], _T] | None*) – If *dir_* is omitted, fetching will result in a `bytes` or a `str`. Callback is a single-argument callable accepting the fetched content and returning anything.

Returns
A tuple with fetched results and the remaining IDs. The former is a list of tuples, where the first element is the original ID, and the second element is either the path to a downloaded file or downloaded data as string. The order may differ. The latter is a list of IDs that failed to fetch.

Return type
tuple[list[tuple[tuple[str, str], *Path* | *_RT* | *_T*]], list[tuple[str, str]]]

abstract property supported_str_formats: list[str]

Returns
A list of formats supported by *fetch_structures()*.

class lXtractor.ext.base.**SupportsAnnotate**(**args*, ***kwargs*)

Bases: Protocol[CT]

A class that serves as basis for annotators – callables accepting a *Chain**-type object and returning a single or multiple objects derived from an initial *Chain**, e.g., via spawn_child <lXtractor.core.chain.Chain.spawn_child().

__init__(**args*, ***kwargs*)

annotate(*c*, **args*, *keep=True*, ***kwargs*)

A method must accept a Chain*-type object and return a single or multiple Chain*-type objects that are the original chain bounds.

Return type
CT | *Iterable*[*CT*]

lXtractor.ext.base.**load_json_callback**(*_*, *res*)

Parameters

- **_** (*Any*) – Arguments to the url_getter() (ignored).
- **res** (*str*) – Fetched string content.

Returns
Parsed json as dict.

Return type
dict

lXtractor.ext.base.**parse_structure_callback**(*inp*, *res*)

Parse the fetched structure.

Parameters

- **inp** (*tuple[str, str]*) – A pair of (id, fmt).
- **res** (*str | bytes*) – The fetching result. By default, if fmt in ["cif", "pdb"], the result is str, while fmt="mmtf" will produce bytes.

Returns
Parse generic structure.

Return type
GenericStructure

lXtractor.ext.hmm module

Wrappers around PyHMMer for convenient annotation of domains and families.

class lXtractor.ext.hmm.**Pfam**(*resource_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtractor/checkouts*
resource_name='Pfam')

Bases: *AbstractResource*

A minimalistic Pfam interface.

- *fetch()* fetches Pfam raw HMM models and associated metadata.
- ***parse()* prepares these data for later usage and stores** to the filesystem.
- *read()* loads parsed files.

Parsed Pfam data is represented as a Pandas DataFrame accessible via *df()* with columns: "ID", "Accession", "Description", "Category", and "HMM". Each row corresponds to a single model from Pfam-A collection and associated metadata taken from the Pfam-A.dat file. HMM models are wrapped into a *PyHMMer* instance.

For quick access to a single HMM model parsed into *PyHMMer*, use `Pfam()[hmm_id]`.

__init__(*resource_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtractor/checkouts/latest/lXtractor/res*
resource_name='Pfam')

Parameters

- **resource_path** (*Path*) – Path to parsed resource data.
- **resource_name** (*str*) – Resource's name.

clean(*raw=True*, *parsed=False*)

Remove Pfam data. If *raw* and *parsed* are both `False`, removes the `path` with all stored data.

Parameters

- **raw** (*bool*) – Remove raw fetched files.
- **parsed** (*bool*) – Remove parsed files.

Returns
Nothing.

Return type
None

dump(*path=None*)

Store parsed data to the filesystem.

This function will store the HMM metadata to attr:*path* / "parsed" / "dat.csv" and separate gzip-compressed HMM models into `path` / "parsed" / "hmm".

Parameters
path (*Path | None*) – Use this path instead of the `path` as a base dir.

Returns
The path `path` / "parsed".

Return type
Path

fetch(*url_hmm='https://ftp.ebi.ac.uk/pub/databases/Pfam/current_release/Pfam-A.hmm.gz'*, *url_dat='https://ftp.ebi.ac.uk/pub/databases/Pfam/current_release/Pfam-A.hmm.dat.gz'*)

Fetch Pfam-A data from InterPro.

Parameters

- **url_hmm** (*str*) – URL to "Pfam-A.hmm.gz".
- **url_dat** (*str*) – URL to "Pfam-A.hmm.dat.gz"

Returns
A pair of filepaths for fetched HMM and dat files.

Return type
tuple[*Path*, *Path*]

load_hmm(*df=None*, *path=None*)

Load HMM models according to accessions in passed *df* and create a column "PyHMMer" with loaded models.

Parameters

- **df** (*DataFrame | None*) – A `DataFrame` having all the :meth:`dat_columns.
- **path** (*Path | None*) – A custom path to the parsed data with an "hmm" subdir.

Returns
A copy of the original `DataFrame` with loaded models.

Return type
DataFrame

parse(*dump=True*, *rm_raw=True*)

Parse fetched raw data into a single pandas `DataFrame`.

Parameters

- **dump** (*bool*) – Dump parsed files to `path` / "raw" dir.
- **rm_raw** (*bool*) – Clean up the raw data once parsing is done.

Returns
A parsed Pfam `DataFrame`. See the class's docs for a list of columns.

Return type
DataFrame

read(*path=None*, *accessions=None*, *categories=None*, *hmm=True*)

Read parsed Pfam data.

First it reads the "dat" file and filters to relevant accessions and/or categories. Then, if *hmm* is `True`, it loads each model and wraps into an *PyHMMer* instance. Otherwise, it loads the HMM metadata. One can explore and filter these data, then load the desired HMM models via *load_hmm()*.

Parameters

- **path** (*Path | None*) – A path to the dir with layout similar to what *dump()* creates.
- **accessions** (*Container[str] | None*) – A list of Pfam accessions following the ".", e.g., `["PF00069", ]`.
- **categories** (*Container[str] | None*) – A list of Pfam categories to filter the accessions to.
- **hmm** (*bool*) – Load HMM models.

Returns
A parsed Pfam `DataFrame`.

Return type
DataFrame

property dat_columns: tuple[str, ...]

property df: DataFrame | None

Returns
Parsed Pfam if *read()* or *parse()* were called. Otherwise, returns `None`.

class lXtractor.ext.hmm.**PyHMMer**(*hmm*, ***kwargs*)

Bases: `object`

A basis pyhmmer interface aimed at domain extraction. It works with a single hmm model and pipeline instance.

The original documentation <https://pyhmmer.readthedocs.io/en/stable/>.

__init__(*hmm*, ***kwargs*)

Parameters

- **hmm** (*HMM | HMMFile | Path | str*) – An `HMMFile` handle or path as string or *Path* object to a file containing a single HMM model. In case of multiple models, only the first one will be taken
- **kwargs** – Passed to `Pipeline`. The *alphabet* argument is derived from the supplied *hmm*.

align(*seqs*)

Align sequences to a profile.

Parameters
seqs (*Iterable[Chain | ChainStructure | ChainSequence | str | tuple[str, str] | DigitalSequence]*) – Sequences to align.

Returns
`TextMSA` with aligned sequences.

Return type
TextMSA

annotate(*objs*, *new_map_name=None*, *min_score=None*, *min_size=None*, *min_cov_hmm=None*, *min_cov_seq=None*, *domain_filter=None*, ***kwargs*)

Annotate provided objects by hits resulting from the HMM search.

An annotation is the creation of a child object via `spawn_child()` method (e.g., `lXtractor.core.chain.ChainSequence.spawn_child()`).

Parameters

- **objs** (*Iterable[Chain | ChainStructure | ChainSequence] | Chain | ChainStructure | ChainSequence*) – A single one or an iterable over *Chain**-type objects.
- **new_map_name** (*str | None*) – A name for a child `ChainSequence` <`lXtractor.core.chain.ChainSequence` to hold the mapping to the hmm numbering.
- **min_score** (*float | None*) – Min hit score.

- **min_size** (*int | None*) – Min hit size.
- **min_cov_hmm** (*float | None*) – Min HMM model coverage – a fraction of mapped / total nodes.
- **min_cov_seq** (*float | None*) – Min coverage of a sequence by the HMM model nodes – a fraction of mapped nodes to the sequence's length.
- **domain_filter** (*Callable[[Domain], bool] | None*) – A callable to filter domain hits.
- **kwargs** – Passed to the *spawn_child* method. **Hint:** if you don't want to keep spawned children, pass `keep=False` here.

Returns
A generator over spawned children yielded sequentially for each input object and valid domain hit.

Return type
Generator[*CT*, None, None]

convert_seq(*obj*)

Parameters
obj (*Any*) – A *Chain*-type object or string or a tuple of (name, _seq). A sequence of this object must be compatible with the alphabet of the HMM model.

Returns
A digitized sequence compatible with PyHMMer.

Return type
DigitalSequence

classmethod from_hmm_collection(*hmm*, ***kwargs*)

Split HMM collection and initialize a *PyHMMer* instance from each HMM model.

Parameters
- **hmm** (*_HmmInpT*) – A path to HMM file, opened HMMFile handle, or parsed HMM.
- **kwargs** – Passed to the class constructor.

Returns
A generator over *PyHMMer* instances created from the provided HMM models.

Return type
abc.Generator[t.Self]

classmethod from_msa(*msa*, *name*, *alphabet*, ***kwargs*)

Create a *PyHMMer* instance from a multiple sequence alignment.

Parameters
- **msa** (*abc.Iterable[tuple[str, str] | str | _ChainT] | lXAlignment*) – An iterable over sequences.
- **name** (*str | bytes*) – The HMM model's name.
- **alphabet** (*Alphabet | str*) – An alphabet to use to build the HMM model. See *digitize_seq()* for available options.
- **kwargs** – Passed to `DigitalMSA` of `PyHMMer` that serves as the basis for creating an HMM model.

Returns
A new *PyHMMer* instance initialized with the HMM model built here.

Return type
t.Self

init_pipeline(***kwargs*)

Parameters
kwargs – Passed to `Pipeline` during initialization.

Returns
Initialized pipeline, also saved to *pipeline*.

Return type
Pipeline

search(*seqs*)

Run the *pipeline* to search for *hmm*.

Parameters
seqs (*Iterable[Chain | ChainStructure | ChainSequence | str | tuple[str, str] | DigitalSequence]*) – Iterable over digital sequences or objects accepted by *convert_seq()*.

Returns
Top hits resulting from the search.

Return type
TopHits

hits_: TopHits | None

Hits resulting from the most recent HMM search

hmm

HMM instance

pipeline: Pipeline

Pipeline to use for HMM searches

lXtractor.ext.hmm.**digitize_seq**(*obj*, *alphabet='amino'*)

Parameters

- **obj** (*Any*) – A *Chain**-type object or string or a tuple of (name, _seq). A sequence of this object must be compatible with the alphabet of the HMM model.
- **alphabet** (*Alphabet | str*) – An alphabet type the sequence corresponds to. Can be an initialized PyHMMer alphabet or a string “amino”, “dna”, or “rna”.

Returns
A digitized sequence compatible with PyHMMer.

Return type
DigitalSequence

lXtractor.ext.hmm.**iter_hmm**(*hmm*)

Iterate over HMM models.

Parameters
hmm (*HMM | HMMFile | Path | str*) – A path to an HMM file, opened `HMMFile` or a stream.

Returns
An iterator over individual HMM models.

Return type
Generator[*HMM*]

lXtractor.ext.pdb_ module

Utilities to interact with the RCSB PDB database.

class lXtractor.ext.pdb_.**PDB**(*max_trials=1*, *num_threads=None*, *verbose=False*)

Bases: *StructureApiBase*

Basic RCSB PDB interface to fetch structures and information.

Example of fetching structures:

```
>>> pdb = PDB()
>>> fetched, failed = pdb.fetch_structures(['2src', '2oiq'], dir_=None)
>>> len(fetched) == 2 and len(failed) == 0
True
>>> (args1, res1), (args2, res2) = fetched
>>> assert {args1, args2} == {('2src', 'cif'), ('2oiq', 'cif')}
>>> isinstance(res1, str) and isinstance(res2, str)
True
```

Example of fetching information:

```
>>> pdb = PDB()
>>> fetched, failed = pdb.fetch_info(
...     'entry', [('2SRC', ), ('2OIQ', )], dir_=None)
>>> len(failed) == 0 and len(fetched) == 2
True
>>> (args1, res1), (args2, res2) = fetched
>>> assert {args1, args2} == {('2SRC', ), ('2OIQ', )}
>>> assert isinstance(res1, dict) and isinstance(res2, dict)
```

Hint: Check `list_services()` to list available info services.

__init__(*max_trials=1*, *num_threads=None*, *verbose=False*)

Parameters

- **url_getters** – A dictionary holding functions constructing urls from provided args.
- **max_trials** (*int*) – Max number of fetching attempts for a given query (PDB ID).
- **num_threads** (*int | None*) – The number of threads to use for parallel requests. If None,will send requests sequentially.
- **verbose** (*bool*) – Display progress bar.

static fetch_obsolete()

Returns
A dict where keys are obsolete PDB IDs and values are replacement PDB IDs or an empty string if no replacement was made.

Return type
dict[str, str]

property supported_str_formats: list[str]

Returns
A list of formats supported by `fetch_structures()`.

lXtractor.ext.pdb_.**filter_by_method**(*pdb_ids*, *pdb=<lXtractor.ext.pdb_.PDB object>*, *method='X-ray'*, *dir_=None*)

See also:

`PDB.fetch_info`

Note: Keys for the info dict are 'rcsb_entry_info' -> 'experimental_method'

Parameters

- **pdb_ids** (*Iterable[str]*) – An iterable over PDB IDs.
- **pdb** (PDB) – Fetcher instance. If not provided, will init with default params.
- **method** (*str*) – Method to match. Must correspond exactly.
- **dir** – Dir to save info "entry" json dumps.

Returns
A list of PDB IDs obtained by desired experimental procedure.

Return type
list[str]

lXtractor.ext.pdb_.**url_getters**()

Returns
A dictionary with {name: getter} where getter is a function accepting string args and returning a valid URL.

Return type
dict[str, UrlGetter]

lXtractor.ext.sifts module

Contains utils allowing to benefit from SIFTS database *UniProt-PDBF;* mapping.

Namely, the *SIFTS* class is build around the file *uniprot_segments_observed.csv.gz*. The latter contains segment-wise mapping between UniProt sequences and continuous corresponding regions in PDB structures, and allows us to:

#. Cross-reference PDB and UniProt databases (e.g., which structures are available for a UniProt "PXXXXXX" accession?) #. Map between sequence numbering schemes.

class lXtractor.ext.sifts.**Mapping**(*id_from*, *id_to*, **args*, ***kwargs*)

Bases: UserDict

A dict subclass with explicit IDs of keys/values sources.

__init__(*id_from*, *id_to*, **args*, ***kwargs*)

Parameters

- **id_from** (*str*) – ID of an objects a mapping is from (keys).
- **id_to** (*str*) – ID of an object a mapping is to (values).
- **args** – passed to dict.
- **kwargs** – passed to dict.

class lXtractor.ext.sifts.**SIFTS**(*resource_path=None*, *resource_name='SIFTS'*, *load_segments=False*, *load_id_mapping=False*)

Bases: *AbstractResource*

A resource to segment-wise and ID mappings between UniProt and PDB.

For a first-time usage, you'll need to call *fetch()* to download and store the "uniprot_segments_observed" dataset.

```
>>> sifts = SIFTS()
>>> path = sifts.fetch()
>>> path.name
'uniprot_segments_observed.csv.gz'
```

Next, *parse()* will process the downloaded file to create and store the table with segments and ID mappings.

(We pass overwrite=True for the doctest to work. It's not needed for the first setup).

```
>>> df, mapping = sifts.parse(store_to_resources=True, overwrite=True)
>>> isinstance(df, pd.DataFrame) and isinstance(mapping, dict)
True
>>> list(df.columns)[:4]
['PDB_Chain', 'PDB', 'Chain', 'UniProt_ID']
>>> list(df.columns)[4:]
['PDB_start', 'PDB_end', 'UniProt_start', 'UniProt_end']
```

Now that we parsed SIFTS segments data, we can use it to map IDs and numberings between UniProt and PDB. Let's reinitalize SIFTS to verify it loads locally stored resources

```
>>> sifts = SIFTS(load_segments=True, load_id_mapping=True)
>>> assert isinstance(sifts.df, pd.DataFrame)
>>> assert isinstance(sifts.id_mapping, dict)
```

SIFTS has three types of mappings stored:

1) Between UniProt and PDB Chains

```
>>> sifts['P12931'][:4]
['1A07:A', '1A07:B', '1A08:A', '1A08:B']
```

2) Between PDB Chains and UniProt IDs

```
>>> sifts['1A07:A']
['P12931']
```

3) Between PDB IDs and PDB Chains

```
>>> sifts['1A07']
['A', 'B']
```

The same types of keys are supported to obtain mappings between the numbering schemes. You'll get a generator yielding mappings from UniProt numbering to the PDB numbering.

In these two cases, we'll get the mappings for each chain.

```
>>> mappings = list(sifts.map_numbering('P12931'))
>>> assert len(mappings) == len(sifts['P12931'])
>>> mappings = list(sifts.map_numbering('1A07'))
>>> assert len(mappings) == len(sifts['1A07']) == 2
```

If we specify the chain, we get a single mapping.

```
>>> m = next(sifts.map_numbering('1A07:A'))
>>> list(m.items())[:2]
[(145, 145), (146, 146)]
```

__init__(*resource_path=None*, *resource_name='SIFTS'*, *load_segments=False*, *load_id_mapping=False*)

Parameters

- **resource_path** (*Path | None*) – a path to a file "uniprot_segments_observed". If not provided, will try finding this file in the `resources` module. If the latter fails will attempt fetching the mapping from the FTP server and storing it in the `resources` for later use.
- **resource_name** (*str*) – the name of the resource.
- **load_segments** (*bool*) – load pre-parsed segment-level mapping
- **load_id_mapping** (*bool*) – load pre-parsed id mapping

dump(*path*, ***kwargs*)

Parameters

- **path** (*Path*) – a valid writable path.
- **kwargs** – passed to `DataFrame.to_csv()` method.

Returns

fetch(*url='ftp://ftp.ebi.ac.uk/pub/databases/msd/sifts/flatfiles/csv/uniprot_segments_observed.csv.gz'*, *overwrite=False*)

Download the resource.

`static load()`

Returns
Loaded segments df and name mapping or `None` if they don't exist.

Return type
tuple[*DataFrame* | None, dict[str, list[str]] | None]

`map_id(x)`

Parameters
x (*str*) – Identifier to map from.

Returns
A list of IDs that *x* maps to.

Return type
list[str] | None

`map_numbering(obj_id)`

Retrieve mappings associated with the `obj_id`. Mapping example:

```
1 -> 2
2 -> 3
3 -> None
4 -> 4
```

Above, a UniProt sequence maps to two segments of a PDB sequence (2-3 and 4). PDB sequence is always considered a subset of a corresponding UniProt sequence. Thus, any "holes" between continuous PDB segments are filled with `None`.

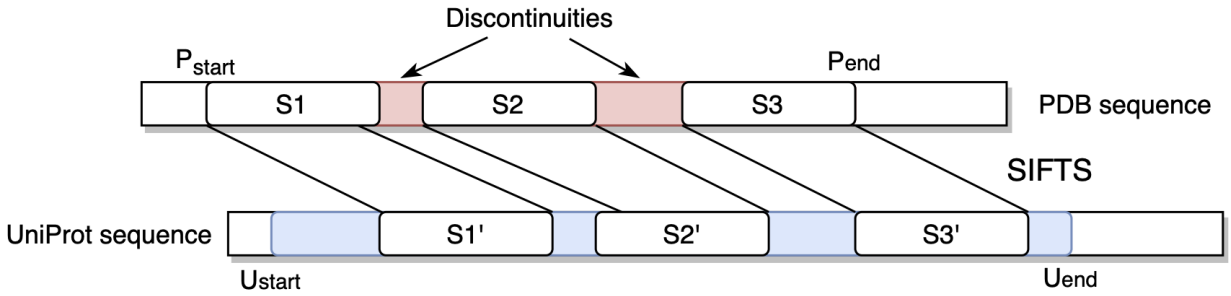

Fig. 1: Mapping from PDB segments to UniProt segments accounting for discontinuities.

See also:

`map_segment_numbering`

wrap_into_segments().

Parameters
obj_id (*str*) – a string value in three possible formats:

1. "PDB ID:Chain ID"
2. "PDB ID"
3. "UniProt ID"

Returns
an iterator over the `Mapping` objects. These are "unidirectional", i.e., the `Mapping` is always from the UniProt numbering to the PDB numbering regardless of the `obj_id` nature.

Return type
Generator[Mapping]

parse(*overwrite=False*, *store_to_resources=True*, *rm_raw=True*)

Prepare the resource to be used for mapping:

- remove records with empty chains.
- **select and rename key columns based on the `SIFTS_RENAMES`** constant.
- create a *PDB_Chain* column to speed up the search.

Parameters

- **`overwrite`** (*bool*) – Overwrite both `df` and existing id mapping and parsed segments.
- **`store_to_resources`** (*bool*) – Store parsed *DataFrame* and id mapping in resources for further simplified access.
- **`rm_raw`** (*bool*) – After parsing is finished, remove raw SIFTS download. **(!) If `store_to_resources` is ``False``, using SIFTS next time will require downloading "uniprot_segments_observed".**

Returns
prepared `DataFrame` of Segment-wise mapping between UniProt and PDB sequences. Mapping between IDs will be stored in `id_mapping`.

Return type
tuple[*DataFrame*, dict[str, list[str]]]

prepare_mapping(*up_ids: Iterable[str]*, *pdb_ids: Iterable[str] | None = None*, *pdb_method: str | None = 'X-ray'*, *pdb_base: Path | None = None*, *pdb_fmt: str = 'cif'*, *pdb_method_filter_kwargs: Mapping[str, Any] | None = None*) → Mapping[str, list[tuple[str | Path, list[str]]]]

prepare_mapping(*up_ids: Mapping[str, _Mkey]*, *pdb_ids: Iterable[str] | None = None*, *pdb_method: str | None = 'X-ray'*, *pdb_base: Path | None = None*, *pdb_fmt: str = 'cif'*, *pdb_method_filter_kwargs: Mapping[str, Any] | None = None*) → Mapping[_Mkey, list[tuple[str | Path, list[str]]]]

prepare_mapping(*up_ids: Iterable[str] | Mapping[str, _Mkey]*, *pdb_ids: Iterable[str] | None = None*, *pdb_method: str | None = 'X-ray'*, *pdb_base: None = None*, *pdb_fmt: str = 'cif'*, *pdb_method_filter_kwargs: Mapping[str, Any] | None = None*) → Mapping[str | _Mkey, list[tuple[str, list[str]]]]

prepare_mapping(*up_ids: Iterable[str] | Mapping[str, _Mkey]*, *pdb_ids: Iterable[str] | None = None*, *pdb_method: str | None = 'X-ray'*, *pdb_base: Path = None*, *pdb_fmt: str = 'cif'*, *pdb_method_filter_kwargs: Mapping[str, Any] | None = None*) → Mapping[str | _Mkey, list[tuple[Path, list[str]]]]

Prepare mapping to use with `lXtractor.core.chain.initializer.ChainInitializer.from_mapping()`.

Uses SIFTS' UniProt-PDB mappings to derive mapping of the form:

```
UniProtID => [(PDB code, [PDB chains]), ...]
```

Parameters

- **up_ids** – UniProt IDs to map with *SIFTS* or a mapping of UniProt IDs to objects allowed as keys in `from_mapping()`.
- **pdb_ids** – PDB IDs to restrict the mapping to. Can be regular IDs or with chain specifier (eg "1ABC:A").
- **pdb_method** – Filter PDB IDs by experimental method.
- **pdb_base** – A path to a PDB files' dir. If provided, the mapping takes the form:

```
UniProtID => [(PDB path, [PDB chains]), ...]
```

- **pdb_fmt** – PDB file format for files in *pdb_base*.
- **pdb_method_filter_kwargs** – A keyword arguments passed to *lXtractor.ext.pdb_.filter_by_method()* used to filter PDB IDs.

Returns

A mapping that is almost ready to be used with `lXtractor.core.chain.initializer.ChainInitializer`. The only preparation step left is to replace the keys with compatible type.

read(*overwrite=True*)

The method reads the initial file "uniprot_segments_observed" into memory.

To load parsed files, use *load()*.

Parameters

overwrite (*bool*) – overwrite existing `df` attribute.

Returns

pandas `DataFrame` object.

Return type

DataFrame

property pdb_chains: set[str]

Returns

A set of encompassed PDB Chains (in {PDB_ID}:{PDB_Chain} format).

property pdb_ids: set[str]

Returns

A set of encompassed PDB IDs.

property uniprot_ids: set[str]

Returns

A set of encompassed UniProt IDs.

lXtractor.ext.sifts.**wrap_into_segments**(*df*)

Parameters

df (*DataFrame*) – A subset of a `Sifts.df` corresponding to a unique "UniProt_ID – PDB_ID:Chain_ID" pair.

Returns

Two lists with the same length (1) UniProt segments, and (2) PDB segments, where segments correspond to each other.

Return type
tuple[list[Segment], list[Segment]]

IXtractor.ext.uniprot module

class lXtractor.ext.uniprot.**UniProt**(*chunk_size=100*, *max_trials=1*, *num_threads=1*, *verbose=False*)

Bases: *ApiBase*

An interface to UniProt fetching.

`UniProt.url_getters` defines functions that construct a URL from provided arguments to fetch specific data. For instance, calling a URL getter for sequences in fasta format using a list of sequences will construct a valid URL for fetching the data.

```
>>> uni = UniProt()
>>> uni.url_getters['sequences'](['P00523', 'P12931'])
'https://rest.uniprot.org/uniprotkb/stream?format=fasta&query=accession
↪%3AP00523+OR+accession%3AP12931'
```

These URLs are constructed dynamically within this class's methods, used to query UniProt, fetch and parse the data.

__init__(*chunk_size=100*, *max_trials=1*, *num_threads=1*, *verbose=False*)

Parameters

- **chunk_size** (*int*) – A number of IDs to join within a single URL and query simultaneously. Note that having invalid URL in a chunk invalidates all its IDs: they won't be fetched. For optimal performance, please filter your accessions carefully.
- **max_trials** (*int*) – A maximum number of trials for fetching a single chunk. Makes sense to raise above 1 when the connection is unstable.
- **num_threads** (*int*) – The number of threads to use for fetching chunks in parallel.
- **verbose** (*bool*) – Display progress bar via stdout.

fetch_info(*accessions*, *fields=None*, *as_df=True*)

Fetch information in tsv format from UniProt.

Parameters

- **accessions** (*Iterable[str]*) – A list of accessions to fetch the info for.
- **fields** (*str | None*) – A comma-separated list of fields to fetch. If `None`, default fields UniProt provides will be used.
- **as_df** (*bool*) – Convert fetched tables into pandas dataframes and join them. Otherwise, return raw text corresponding to each chunk of *accessions*.

Returns
A list of texts per chunk or a single data frame.

Return type
DataFrame | list[str]

fetch_sequences(*accessions*, *dir_*, *overwrite*, *callback: None*) → Iterator[tuple[str, str]]

fetch_sequences(*accessions*, *dir_*, *overwrite*, *callback: Callable[[tuple[str, str]], T]*) → Iterator[T]

Fetch sequences in "fasta" format from UniProt.

Parameters

- **accessions** – A list of valid accessions to fetch.
- **dir** – A directory where individual sequence will be stored. If exists, will filter accessions before fetching unless *overwrite* is `True`.
- **overwrite** – Overwrite existing sequences if they exist in *dir_*.
- **callback** – A function accepting a single sequence and returning anything else. Can be useful to convert sequences into, eg, :class:~lXtractor.chain.sequence.ChainSequence` (for this, pass :meth:~lXtractor.chain.sequence.ChainSequence.from_tuple` here).

Returns

An iterator over fetched sequences (or whatever `callback` returns).

lXtractor.ext.uniprot.**fetch_uniprot**(*acc*, *fmt='fasta'*, *chunk_size=100*, *fields=None*, ***kwargs*)

An interface to the UniProt's search.

Base URL: https://rest.uniprot.org/uniprotkb/stream

Available DB identifiers: See *bioservices <https://bioservices.readthedocs.io/en/main/_modules/bioservices/uniprot.html>*

Parameters

- **acc** (*Iterable[str]*) – an iterable over UniProt accessions.
- **fmt** (*str*) – download format (e.g., "fasta", "gff", "tab", ...).
- **chunk_size** (*int*) – how many accessions to download in a chunk.
- **fields** (*str | None*) – if the `fmt` is "tsv", must be provided to specify which data columns to fetch.
- **kwargs** – passed to *fetch_chunks()*.

Returns

the 'utf-8' encoded results as a single chunk of text.

Return type

str

lXtractor.ext.uniprot.**make_url**(*accessions*, *fmt*, *fields*)

Return type

str

lXtractor.ext.uniprot.**url_getters**()

Return type

dict[str, *Callable*[[...], str]]

3.1.4 IXtractor.util package

IXtractor.util.io module

Various utilities for IO.

lXtractor.util.io.**fetch_chunks**(*it*, *fetcher*, *chunk_size=100*, ***kwargs*)

A wrapper for fetching multiple links with `ThreadPoolExecutor`.

Parameters

- **it** (*Iterable[V]*) – Iterable over some objects accepted by the *fetcher*, e.g., links.
- **fetcher** (*Callable[[list[V]], T]*) – A callable accepting a chunk of objects from *it*, fetching and returning the result.
- **chunk_size** (*int*) – Split iterable into this many chunks for the executor.
- **kwargs** – Passed to *fetch_iterable()*.

Returns
A list of results

Return type
Generator[tuple[list[*V*], *T* | *Future*], None, None]

lXtractor.util.io.**fetch_iterable**(*it*, *fetcher*, *num_threads=None*, *verbose=False*, *blocking=True*, *allow_failure=True*)

Parameters

- **it** (*Iterable[V]*) – Iterable over some objects accepted by the *fetcher*, e.g., links.
- **fetcher** (*Callable[[V], T]*) – A callable accepting a chunk of objects from *it*, fetching and returning the result.
- **num_threads** (*int | None*) – The number of threads for `ThreadPoolExecutor`.
- **verbose** (*bool*) – Enable progress bar and warnings/exceptions on fetching failures.
- **blocking** (*bool*) – If `True`, will wait for each result. Otherwise, will return `Future` objects instead of fetched data.
- **allow_failure** (*bool*) – If `True`, failure to fetch will raise a warning isntead of an exception. Otherwise, the warning is logged, and the results won't contain inputs that failed to fetch.

Returns
A list of tuples where the first object is the input and the second object is the fetched data.

Return type
Generator[tuple[*V*, *T*], None, None] | *Generator*[tuple[*V*, *Future*[*T*]], None, None]

lXtractor.util.io.**fetch_text**(*url*, *decode=False*, *chunk_size=8192*, ***kwargs*)

Fetch the content as a single string. This will use the `requests.get` with `stream=True` by default to split the download into chunks and thus avoid taking too much memory at once.

Parameters

- **url** (*str*) – Link to fetch from.
- **decode** (*bool*) – Decode the received bytes to utf-8.

- **chunk_size** (*int*) – The number of bytes to use when splitting the fetched result into chunks.
- **kwargs** – Passed to `requests.get()`.

Returns
Fetched text as a single string.

Return type
str | bytes

`lXtractor.util.io.`**fetch_to_file**(*url*, *fpath=None*, *fname=None*, *root_dir=None*, *decode=False*)

Parameters

- **url** (*str*) – Link to a file.
- **fpath** (*Path | None*) – Path to a file for saving. If provided, *fname* and *root_dir* are ignored. Otherwise, will use `.../{this}` from the link for the file name and save into the current dir.
- **fname** (*str | None*) – Name of the file to save.
- **root_dir** (*Path | None*) – Dir where to save the file.
- **decode** (*bool*) – If `True`, try decoding the raw request's content.

Returns
Local path to the file.

Return type
Path

`lXtractor.util.io.`**fetch_urls**(*url_getter*, *url_getter_args*, *fmt*, *dir_*, ***, *fname_idx=0*, *args_applier=None*, *callback=None*, *overwrite=False*, *decode=False*, *max_trials=1*, *num_threads=None*, *verbose=False*)

A general-purpose function for fetching URLs. Each URL is dynamically produced via URL getters supplied with positional arguments.

See also:

ApiBase or PDB for more information on URL getters.

It has two modes: fetching to text and fetching to files. The former is the default, whereas the latter can be turned on by providing *dir_* argument. If provided, each url is considered a separate file to fetch. Thus, the function will also check *dir_* (if it exists) for files that were already fetched to avoid useless work. This can be turned off via *overwrite=True*. For this functionality to work, each argument in *url_getter_args* must be converted to a single (file)name. If an argument is a sequence, *fname_idx* should point to an index, such that `arg[fname_idx]` is the filename.

Parameters

- **url_getter** (*UrlGetter*) – A callable accepting two or more strings and returning a valid url to fetch. The last argument is reserved for *fmt*.
- **url_getter_args** (*Iterable[_U]*) – An iterable over strings or tuple of strings supplied to the *url_getter*. Each element must be sufficient for the *url_getter* to return a valid URL.
- **dir** – Dir to save files to. If `None`, will return either raw string or json-derived dictionary if the *fmt* is "json".
- **fmt** (*str*) – File format. It is used construct a full file name "{filename}.{fmt}".

- **fname_idx** (*int*) – If an element in *url_getter_args* is a tuple, this argument is used to index this tuple to construct a file name that is used to save file / check if such file exists.
- **args_applier** (*Callable[[UrlGetter, _U], str] | None*) – A callable accepting a URL getter and its args and applying the arguments to the URL getter to obtain the URL. If none, will apply arguments as positional arguments.
- **callback** (*Callable[[_U, str | bytes], T] | None*) – A callable to parse content right after fetching, e.g., `json.loads`. It's only used if *dir_* is not provided.
- **overwrite** (*bool*) – Overwrite existing files if *dir_* is provided.
- **decode** (*bool*) – Decode the fetched content (bytes to utf-8). Should be `True` if expecting text content.
- **max_trials** (*int*) – Max number of fetching attempts for a given id.
- **num_threads** (*int | None*) – The number of threads to use for parallel requests. If `None`, will send requests sequentially.
- **verbose** (*bool*) – Display progress bar.

Returns
A tuple with fetched results and the remaining file names. The former is a list of tuples, where the first element is the original name, and the second element is either the path to a downloaded file or downloaded data as string. The order may differ. The latter is a list of names that failed to fetch.

Return type
tuple[list[tuple[*_U*, *_F*] | tuple[*_U*, *T*]], list[*_U*]]

lXtractor.util.io.**get_dirs**(*path*)

Parameters
path (*Path*) – Path to a directory.

Returns
Mapping {dir name => dir path} for each dir in *path*.

Return type
dict[str, *Path*]

lXtractor.util.io.**get_files**(*path*)

Parameters
path (*Path*) – Path to a directory.

Returns
Mapping {file name => file path} for each file in *path*.

Return type
dict[str, *Path*]

lXtractor.util.io.**parse_suffix**(*path*)

Parse a file suffix.

1. If there are no suffixes: raise an error.
2. If there is one suffix, return it.
3. If there are more than one suffixes, join the last two and return.

Parameters
path (*Path*) – Input path.

Returns
Parsed suffix.

Raises
FormatError – If not suffix is present.

Return type
str

lXtractor.util.io.**path_tree**(*path*)

Create a tree graph from Chain*-type objects saved to the filesystem.

The function will recursively walk starting from the provided path, connecting parent and children paths (residing within "segments" directory). If it meets a path containing "structures" directory, it will save valid structure paths under a node's "structures" attribute. In that case, such structures are assumed to be nested under a chain, and they do not form nodes in this graph.

A path to a Chain*-type object is valid if it contains "sequence.tsv" and "meta.tsv" files. A valid structure path must contain "sequence.tsv", "meta.tsv", and "structure.*" files.

Parameters
path (*Path*) – A root path to start with.

Returns
An undirected graph with paths as nodes and edges representing parent-child relationships.

Return type
DiGraph

lXtractor.util.io.**read_n_col_table**(*path*, *n*, *sep='\t'*)

Read table from file and ensure it has exactly *n* columns.

Return type
DataFrame | None

lXtractor.util.io.**run_sp**(*cmd*, *split=True*)

It will attempt to run the command as a subprocess returning text. If the command returns *CalledProcessError*, it will rerun the command with `check=False` to capture all the outputs into the result.

Parameters

- **cmd** (*str*) – A single string of a command.
- **split** (*bool*) – Split *cmd* before running. If `False`, will pass `shell=True`.

Returns
Result of a subprocess with captured output.

lXtractor.util.misc module

Miscellaneous utilities that couldn't be properly categorized.

lXtractor.util.misc.**all_logging_disabled**(*highest_level=50*)

A context manager that will prevent any logging messages triggered during the body from being processed.

The function was borrowed from this gist

Parameters
highest_level – the maximum logging level in use. This would only need to be changed if a custom level greater than CRITICAL is defined.

lXtractor.util.misc.**apply**(*fn*, *it*, *verbose*, *desc*, *num_proc*, *total=None*, *use_joblib=False*, ***kwargs*)

Parameters

- **fn** (*Callable[[T], R]*) – A one-argument function.
- **it** (*Iterable[T]*) – An iterable over some objects.
- **verbose** (*bool*) – Display progress bar.
- **desc** (*str*) – Progress bar description.
- **num_proc** (*int*) – The number of processes to use. Anything below 1 indicates sequential processing. Otherwise, will apply `fn` in parallel using `ProcessPoolExecutor`.
- **total** (*int | None*) – The total number of elements. Used for the progress bar.
- **use_joblib** (*bool*) – Use `joblib.Parallel` for parallel application.

Returns
Passed to `ProcessPoolExecutor.map()` or `joblib.Parallel`.

Return type
Iterator[*R*]

lXtractor.util.misc.**col2col**(*df*, *col_fr*, *col_to*)

Parameters

- **df** (*DataFrame*) – Some DataFrame.
- **col_fr** (*str*) – A column name to map from.
- **col_to** (*str*) – A column name to map to.

Returns
Mapping between values of a pair of columns.

lXtractor.util.misc.**get_cpu_count**(*c*)

lXtractor.util.misc.**graph_reindex_nodes**(*g*)

Reindex the graph nodes so that node data equals to node indices.

Parameters
g (*PyGraph*) – An arbitrary PyGraph.

Returns
A PyGraph of the same size and having the same edges but with reindexed nodes.

Return type
PyGraph

lXtractor.util.misc.**is_empty**(*x*)

Return type
bool

lXtractor.util.misc.**is_valid_field_name**(*s*)

Parameters
s (*str*) – Some string.

Returns
`True if s` is a valid field name for ``__getattr__ `` operations else ``False`.

Return type
bool

lXtractor.util.misc.**json_to_molgraph**(*inp*)

Converts a JSON-formatted molecular graph into a PyGraph object. This graph is a dictionary with two keys: "num_nodes" and "edges". The former indicates the number of atoms in a structure, whereas the latter is a list of edge tuples.

Parameters
inp (*dict | PathLike*) – A dictionary or a path to a JSON file produced using *rustworkx.node_link_json*.

Returns
A graph with nodes and edges initialized in order given in *inp*. Any associated data will be omitted.

Return type
PyGraph

lXtractor.util.misc.**valgroup**(*m*, *sep=':'*)

Reformat a mapping from the format:

```
X => [Y{sep}Z, ...]
```

To a format:

```
X => [(Y, [Z, ...]), ...]
```

```
>>> mapping = {'X': ['C:A', 'C:B', 'Y:Z']}
>>> valgroup(mapping)
{'X': [('X', ['A', 'B']), ('Y', ['Z'])]}
```

Hint: This method is useful for converting the sequence-to-structure mapping outputted by *lXtractor.ext.sifts.SIFTS* to a format accepted by the **:method:`lXtractor.core.chain.initializer.ChainInitializer.from_mapping`** to initialize lXtractor.core.chain.Chain objects

---

Parameters

- **m** (*Mapping[str, list[str]]*) – A mapping from strings to a list of strings.
- **sep** (*str*) – A separator of each mapped string in the list.

Returns
A reformatted mapping.

lXtractor.util.seq module

Low-level utilities to work with sequences (as strings) or sequence files.

lXtractor.util.seq.**biotite_align**(*seqs*, ***kwargs*)

Align two sequences using biotite *align_optimal* function.

Parameters

- **seqs** (*Iterable[tuple[str, str]]*) – An iterable with exactly two sequences.
- **kwargs** – Additional arguments to *align_optimal*.

Returns
A pair of aligned sequences.

Return type
tuple[tuple[str, str], tuple[str, str]]

lXtractor.util.seq.**mafft_add**(*msa*, *seqs*, *, *mafft='mafft'*, *thread=1*, *keeplength=True*)

Add sequences to existing MSA using mafft.

This is a curried function: incomplete argument set yield partially evaluated function (e.g., `mafft_add(thread=10)`).

Parameters

- **msa** (*Iterable[tuple[str, str]] | Path*) – an iterable over sequences with the same length.
- **seqs** (*Iterable[tuple[str, str]]*) – an iterable over sequences comprising the addition.
- **thread** (*int*) – how many threads to dedicate for *mafft*.
- **keeplength** (*bool*) – force to preserve the MSA's length.
- **mafft** (*str*) – *mafft* executable.

Returns
A tuple of two lists of *SeqRecord* objects: with (1) alignment sequences with addition, and (2) aligned addition, separately.

Return type
Iterator[tuple[str, str]]

lXtractor.util.seq.**mafft_align**(*seqs*, *, *mafft='mafft-linsi'*, *thread=1*)

Align an arbitrary number of sequences using mafft.

Parameters

- **seqs** (*Iterable[tuple[str, str]] | Path*) – An iterable over (header, _seq) pairs or path to file with sequences to align.
- **thread** (*int*) – How many threads to dedicate for *mafft*.
- **mafft** (*str*) – *mafft* executable (path or env variable).

Returns
An Iterator over aligned (header, _seq) pairs.

Return type
Iterator[tuple[str, str]]

lXtractor.util.seq.**map_pairs_numbering**(*s1*, *s1_numbering*, *s2*, *s2_numbering*, *align=True*, *align_method=<function mafft_align>*, *empty=None*, ***kwargs*)

Map numbering between a pair of sequences.

Parameters

- **s1** (*str*) – The first sequence.
- **s1_numbering** (*Iterable[int]*) – The first sequence's numbering.
- **s2** (*str*) – The second sequence.
- **s2_numbering** (*Iterable[int]*) – The second sequence's numbering.
- **align** (*bool*) – Align before calculating. If `False`, sequences are assumed to be aligned.
- **align_method** (`AlignMethod`) – Align method to use. Must be a callable accepting and returning a list of sequences.
- **empty** (*Any | None*) – Empty numeration element in place of a gap.
- **kwargs** – Passed to *align_method*.

Returns

Iterator over character pairs (*a*, *b*), where *a* and *b* are the original sequences' numberings. One of *a* or *b* in a pair can be *empty* to represent a gap.

Return type

Generator[tuple[int | None, int | None], None, None]

lXtractor.util.seq.**partition_gap_sequences**(*seqs*, *max_fraction_of_gaps=1.0*)

Removes sequences having fraction of gaps above the given threshold.

Parameters

- **seqs** (*Iterable[tuple[str, str]]*) – a collection of arbitrary sequences.
- **max_fraction_of_gaps** (*float*) – a threshold specifying an upper bound on allowed fraction of gap characters within a sequence.

Returns

a filtered list of sequences.

Return type

tuple[*Iterator*[str], *Iterator*[str]]

lXtractor.util.seq.**read_fasta**(*inp*, *strip_id=True*)

Simple lazy fasta reader.

Parameters

- **inp** (*str | PathLike | TextIOBase | Iterable[str]*) – Pathlike object compatible with `open` or opened file or an iterable over lines or raw text as str.
- **strip_id** (*bool*) – Strip ID to the first consecutive (spaceless) string.

Returns

An iterator of (header, seq) pairs.

Return type

Iterator[tuple[str, str]]

lXtractor.util.seq.**remove_gap_columns**(*seqs*, *max_gaps=1.0*)

Remove gap columns from a collection of sequences.

Parameters

- **seqs** (*Iterable[str]*) – A collection of equal length sequences.
- **max_gaps** (*float*) – Max fraction of gaps allowed per column.

Returns
Initial seqs with gap columns removed and removed columns' indices.

Return type
tuple[*Iterator*[str], *ndarray*]

lXtractor.util.seq.**write_fasta**(*inp*, *out*)

Simple fasta writer.

Parameters

- **inp** (*Iterable[tuple[str, str]]*) – Iterable over (header, _seq) pairs.
- **out** (*Path | SupportsWrite*) – Something that supports *.write* method.

Returns
Nothing.

Return type
None

lXtractor.util.structure module

Low-level utilities to work with structures.

lXtractor.util.structure.**calculate_dihedral**(*atom1*, *atom2*, *atom3*, *atom4*)

Calculate angle between planes formed by [a1, a2, atom3] and [a2, atom3, atom4].

Each atom is an array of shape (3,) with XYZ coordinates.

Calculation method inspired by https://math.stackexchange.com/questions/47059/how-do-i-calculate-a-dihedral-angle-given-cartesian-coordinates

Return type
float

lXtractor.util.structure.**compare_arrays**(*a*, *b*, *eps=0.001*)

Compare two numerical arrays.

Parameters

- **a** (*ndarray[Any, dtype[float | int]]*) – The first array.
- **b** (*ndarray[Any, dtype[float | int]]*) – The second array.
- **eps** (*float*) – Comparison tolerance.

Returns
`True` if the absolute difference between the two arrays is within *eps*.

Raises
LengthMismatch – If the two arrays are not of the same shape.

lXtractor.util.structure.**compare_coord**(*a*, *b*, *eps=0.001*)

Compare coordinates between atoms of two atom arrays.

Parameters

- **a** (*AtomArray*) – The first atom array.
- **b** (*AtomArray*) – The second atom array.
- **eps** (*float*) – Comparison tolerance.

Returns

True if the two arrays are of the same length and the absolute difference between coordinates of the corresponding atom pairs is within *eps*.

lXtractor.util.structure.**extend_residue_mask**(*a*, *idx*)

Extend a residue mask for given atoms.

Parameters

- **a** (*AtomArray*) – An arbitrary atom array.
- **idx** (*list[int]*) – Indices pointing to atoms at which to extend the mask.

Returns

The extended mask, where True indicates that the atom belongs to the same residue as indicated by *idx*.

Return type

ndarray[*Any*, *dtype*[*bool_*]]

lXtractor.util.structure.**filter_any_polymer**(*a*, *min_size=2*)

Get a mask indicating atoms being a part of a macromolecular polymer: peptide, nucleotide, or carbohydrate.

Parameters

- **a** (*AtomArray*) – Array of atoms.
- **min_size** (*int*) – Min number of polymer monomers.

Returns

A boolean mask True for polymers' atoms.

Return type

ndarray

lXtractor.util.structure.**filter_ligand**(*a*)

Filter for ligand atoms – non-polymer and non-solvent hetero atoms.

..note ::

No contact-based verification is performed here.

Parameters

a (*AtomArray*) – Atom array.

Returns

A boolean mask True for ligand atoms.

Return type

ndarray

lXtractor.util.structure.**filter_polymer**(*a*, *min_size=2*, *pol_type='peptide'*)

Filter for atoms that are a part of a consecutive standard macromolecular polymer entity.

Parameters

- **a** (*AtomArray*) – The array to filter.
- **min_size** – The minimum number of monomers.
- **pol_type** – The polymer type, either "peptide", "nucleotide", or "carbohydrate". Abbreviations are supported: "p", "pep", "n", etc.

Returns

This array is *True* for all indices in *array*, where atoms belong to consecutive polymer entity having at least *min_size* monomers.

Return type

ndarray[*Any*, *dtype*[*bool_*]]

lXtractor.util.structure.**filter_selection**(*array*, *res_id*, *atom_names=None*)

Filter AtomArray by residue numbers and atom names.

Parameters

- **array** (*AtomArray*) – Arbitrary structure.
- **res_id** (*Sequence[int] | None*) – A sequence of residue numbers.
- **atom_names** (*Sequence[Sequence[str]] | Sequence[str] | None*) – A sequence of atom names (broadcasted to each position in *res_id*) or an iterable over such sequences for each position in *res_id*.

Returns

A binary mask that is True for filtered atoms.

Return type

ndarray

lXtractor.util.structure.**filter_solvent_extended**(*a*)

Filter for solvent atoms using a curated solvent list including non-water molecules typically being a part of a crystallization solution.

Parameters

a (*AtomArray*) – Atom array.

Returns

A boolean mask True for solvent atoms.

Return type

ndarray

lXtractor.util.structure.**filter_to_common_atoms**(*a1*, *a2*, *allow_residue_mismatch=False*)

Filter to atoms common between residues of atom arrays a1 and a2.

Parameters

- **a1** (*AtomArray*) – Arbitrary atom array.
- **a2** (*AtomArray*) – Arbitrary atom array.
- **allow_residue_mismatch** (*bool*) – If True, when residue names mismatch, the common atoms are derived from the intersection a1.atoms & a2.atoms & {"C", "N", "CA", "CB"}.

Returns
A pair of masks for a1 and a2, `True` for matching atoms.

Raises
`ValueError` –

1. If *a1* and *a2* have different number of residues.
2. **If the selection for some residue produces different number** of atoms.

Return type
tuple[*ndarray*, *ndarray*]

`lXtractor.util.structure.`**`find_contacts`**(*a*, *mask*)

Find contacts between a subset of atoms within the structure and the rest of the structure. An atom is considered to be in contact with another atom if the distance between them is below the threshold for the non-covalent bond specified in config (`DefaultConfig["bonds"]["NC-NC"][1]`).

Parameters

- **`a`** (*`AtomArray`*) – Atom array.
- **`mask`** (*`ndarray`*) – A boolean mask `True` for atoms for which to find contacts.

Returns

A tuple with three arrays of size equal to the *a*'s number of atoms:

1. **Contact mask: `True` for `a[~mask]` atoms in contact with** `a[mask]`.
2. Distances: for `a[mask]` atoms to the closest `a[~mask]` atom.
3. Indices: of these closest `a[~mask]` atoms within the *mask*.

Suppose that `mask` specifies a ligand. Then, for `i`-th atom in *a*, `contacts[i]`, `distances[i]`, `indices[i]` indicate whether `a[i]` has a contact, the precise distance from `a[i]` atom to the closest ligand atom, and an index of this ligand atom, respectively.

Return type
tuple[*ndarray*, *ndarray*, *ndarray*]

`lXtractor.util.structure.`**`find_first_polymer_type`**(*a*, *min_size=2*, *order=('p', 'n', 'c')*)

Determines polymer type of the supplied atom array or an array of atom marks.

Probe polymer types in a sequence in a given order. If a polymer with at least *min_size* atoms of the probed type is found, it will be returned.

Hint: The function serves as a good quick-check when a single polymer type is expected, which should always be true when *a* is an array of atom marks.

Parameters

- **`a`** (*`AtomArray | ndarray[Any, dtype[int]]`*) – An arbitrary array of atoms.
- **`min_size`** (*`int`*) – A minimum number of monomers in a polymer.
- **`order`** (*`tuple[str, str, str]`*) – An order of the polymers to probe.

Returns
The first polymer type to accommodate *min_size* requirement.

Return type
str

lXtractor.util.structure.**find_primary_polymer_type**(*a*, *min_size=2*, *residues=False*)

Find the major polymer type, i.e., the one with the largest number of atoms or monomers.

Parameters

- **a** (*AtomArray*) – An arbitrary atom array.
- **min_size** (*int*) – Minimum number of monomers for a polymer.
- **residues** (*bool*) – `True` if the dominant polymer should be picked according to the number of residues. Otherwise, the number of atoms will be used.

Returns
A binary mask pointing at the polymer atoms in *a* and the polymer type – "c" (carbohydrate), "n" (nucleotide), or "p" (peptide). If no polymer atoms were found, polymer type will be designated as "x".

Return type
tuple[*ndarray*, str]

lXtractor.util.structure.**get_missing_atoms**(*a*, *excluding_names=('OXT',)*, *excluding_elements=('H',)*)

For each residue, compare with the one stored in CCD, and find missing atoms.

Parameters

- **a** (*AtomArray*) – Non-empty atom array.
- **excluding_names** (*Sequence[str] | None*) – A sequence of atom names to exclude for calculation.
- **excluding_elements** (*Sequence[str] | None*) – A sequence of element names to exclude for calculation.

Returns
A generator of lists of missing atoms (excluding hydrogens) per residue in *a* or `None` if not such residue was found in CCD.

Return type
Generator[list[str | None] | None, None, None]

lXtractor.util.structure.**get_observed_atoms_frac**(*a*, *excluding_names=('OXT',)*, *excluding_elements=('H',)*)

Find fractions of observed atoms compared to canonical residue versions stored in CCD.

Parameters

- **a** (*AtomArray*) – Non-empty atom array.
- **excluding_names** (*Sequence[str] | None*) – A sequence of atom names to exclude for calculation.
- **excluding_elements** (*Sequence[str] | None*) – A sequence of element names to exclude for calculation.

Returns
A generator of observed atom fractions per residue in *a* or `None` if a residue was not found in CCD.

Return type
Generator[list[str | None] | None, None, None]

lXtractor.util.structure.**iter_canonical**(*a*)

Parameters
a (*AtomArray*) – Arbitrary atom array.

Returns
Generator of canonical versions of residues in *a* or `None` if no such residue found in CCD.

Return type
Generator[*AtomArray* | None, None, None]

lXtractor.util.structure.**iter_residue_masks**(*a*)

Iterate over residue masks.

Parameters
a (*AtomArray*) – Atom array.

Returns
A generator over boolean masks for each residue in *a*.

Return type
Generator[*ndarray*[*Any*, *dtype*[*bool_*]], None, None]

lXtractor.util.structure.**load_structure**(*inp*, *fmt=''*, ***, *gz=False*, ***kwargs*)

This is a simplified version of a `biotite.io.general.load_structure` extending the supported input types. Namely, it allows using paths, strings, bytes or gzipped files. On the other hand, there are less supported formats: pdb, cif, and mmtf.

Parameters

- **inp** (*IOBase | Path | str | bytes*) – Input to load from. It can be a path to a file, an opened file handle, a string or bytes of file contents. Gzipped bytes and files are supported.
- **fmt** (*str*) – If `inp` is a `Path`-like object, it must be of the form "name.fmt" or "name.fmt.gz". In this case, `fmt` is ignored. Otherwise, it is used to determine the parser type and must be provided.
- **gz** (*bool*) – If `inp` is gzipped `bytes`, this flag must be `True`.
- **kwargs** – Passed to `get_structure`: either a method or a separate function used by `biotite` to convert the input into an `AtomArray`.

Returns

Return type
AtomArray

lXtractor.util.structure.**mark_polymer_type**(*a*, *min_size=2*)

Denote polymer type in an atom array.

It will find the breakpoints in *a* and split it into segments. Each segment will be checked separately to determine its polymer type. The results are then concatenated into a single array and returned.

Parameters

- **a** (*AtomArray*) – Any atom array.
- **min_size** (*int*) – Minimum number of consecutive monomers in a polymer.

Returns
An array where each atom of *a* is marked by a character: `"n"`, `"p"`, or `"c"` for nucleotide, peptide, and carbohydrate. Non-polymer atoms are marked by "x".

Return type
ndarray[*Any*, *dtype*[*str_*]]

lXtractor.util.structure.**save_structure**(*array*, *path*, ***kwargs*)

This is a simplified version of a `biotite.io.general.save_structure`. On the one hand, it can conveniently compress the data using `gzip`. On the other hand, the number of supported formats is fewer: pdb, cif, and mmtf.

Parameters

- **array** (*AtomArray*) – An `AtomArray` to write.
- **path** (*Path*) – A path with correct extension, e.g., `Path("data/structure.pdb")`, or `Path("data/structure.pdb.gz")`.
- **kwargs** – If compressing is not required, the original `save_structure` from biotite is used with these `kwargs`. Otherwise, `kwargs` are ignored.

Returns
If the file was successfully written, returns the original *path*.

lXtractor.util.structure.**to_graph**(*a*, *split_chains=False*)

Create a molecular connectivity graph from an atom array.

Molecular graph is a undirected graph without multiedges, where nodes are indices to atoms. Thus, node indices point directly to atoms in the provided atom array, and the number of nodes equals the number of atoms. A pair of nodes has an edge between them, if they form a covalent bond. The edges are constructed according to atom-depended bond thresholds defined by the global config. These distances are stored as edge values. See the docs of *rustworkx* on how to manipulate the resulting graph object.

Parameters

- **a** (*AtomArray*) – Atom array to guild a graph from.
- **split_chains** (*bool*) – Edges between atoms from different chains are forbidden.

Returns
A graph object where nodes are atom indices and edges represent covalent bonds.

Return type
PyGraph

3.1.5 lXtractor.variables package

lXtractor.variables.base module

Base classes, common types and functions for the *variables* module.

class lXtractor.variables.base.**AbstractCalculator**

Bases: `Generic[OT]`

Class defining variables' calculation strategy.

abstract **__call__**(*o: OT*, *v: VT*, *m: Mapping[int, int | None] | None*) → tuple[bool, RT]

abstract **__call__**(*o: Iterable[OT]*, *v: Iterable[VT] | Iterable[Iterable[VT]]*, *m: Iterable[Mapping[int, int | None] | None] | None*) → Iterable[Iterable[tuple[bool, RT]]]

Parameters

- **o** – Object to calculate on.
- **v** – Some variable whose *calculate* method accepts *o*-type instances.

- **m** – Optional mapping between object and some reference object numbering schemes.

Returns
Calculation result.

abstract map(*o*, *v*, *m*)

Map variables to a single object.

Parameters

- **o** (*OT*) – Object to calculate on.
- **v** (*Iterable[VT]*) – An iterable over variables whose *calculate* method accepts *o*-type instances.
- **m** (*Mapping[int, int | None] | None*) – Optional mapping between object and some reference object numbering schemes.

Returns
An iterator (generator) over calculation result.

Return type
Iterable[tuple[bool, *RT*]]

abstract vmap(*o*, *v*, *m*)

Map objects to a single variable.

Parameters

- **o** (*Iterable[OT]*) – An iterable over objects to calculate on.
- **v** (*VT*) – Some variable whose *calculate* method accepts *o*-type instances.
- **m** (*Iterable[Mapping[int, int | None] | None]*) – Optional mapping between object and some reference object numbering schemes.

Returns
An iterator (generator) over calculation result.

Return type
Iterable[tuple[bool, *RT*]]

class lXtractor.variables.base.**AbstractVariable**

Bases: Generic[OT, RT]

Abstract base class for variables.

abstract calculate(*obj*, *mapping=None*)

Calculate variable. Each variable defines its own calculation strategy.

Parameters

- **obj** (*OT*) – An object used for variable's calculation.
- **mapping** (*Mapping[int, int | None] | None*) – Mapping from generalizable positions of MSA/reference/etc. to the *obj*'s positions.

Returns
Calculation result.

Raises
FailedCalculation if the calculation fails.

Return type
RT

property id: str

Variable identifier such that eval(x.id) produces another instance.

abstract property rtype: Type[RT]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.base.**AggFn**(**args*, ***kwargs*)

Bases: Protocol

__call__(*a*, ***kwargs*)

Call self as a function.

Return type
ndarray | float

__init__(**args*, ***kwargs*)

class lXtractor.variables.base.**LigandVariable**

Bases: *AbstractVariable*[*Ligand*, RT], Generic[T, RT]

A type of variable whose *calculate()* method requires protein sequence.

abstract calculate(*obj*, *mapping=None*)

Parameters

- **obj** (Ligand) – Some sequence.
- **mapping** (*Mapping[int, int | None] | None*) – Optional mapping between sequence and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
RT

class lXtractor.variables.base.**ProtFP**(*path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtractor/checkou*

Bases: object

ProtFP embeddings for amino acid residues.

ProtFP is a coding scheme derived from the PCA analysis of the AAIndex database [Westen *et al.*, 2013, Westen *et al.*, 2013].

```
>>> pfp = ProtFP()
>>> pfp[('G', 1)]
-5.7
>>> list(pfp['G'])
[-5.7, -8.72, 4.18, -1.35, -0.31]
>>> comp1 = pfp[1]
>>> assert len(comp1) == 20
>>> comp1[0]
-5.7
>>> comp1.index[0]
'G'
```

__init__(*path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lxtractor/checkouts/latest/lXtractor/resources/PF*

class lXtractor.variables.base.**SequenceVariable**

Bases: *AbstractVariable*[Sequence[T], RT], Generic[T, RT]

A type of variable whose *calculate()* method requires protein sequence.

abstract **calculate**(*obj*, *mapping=None*)

Parameters

- **obj** (*Sequence[T]*) – Some sequence.
- **mapping** (*Mapping[int, int | None] | None*) – Optional mapping between sequence and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
RT

class lXtractor.variables.base.**StructureVariable**

Bases: *AbstractVariable*[*GenericStructure*, RT], Generic[RT]

A type of variable whose *calculate()* method requires protein structure.

abstract **calculate**(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*Mapping[int, int | None] | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
RT

class lXtractor.variables.base.**Variables**(*dict=None*, */*, ***kwargs*)

Bases: UserDict

A subclass of dict holding variables (*AbstractVariable* subclasses).

The keys are the *AbstractVariable* subclasses' instances (hashed by :meth:*id*), and values are calculation results.

as_df()

Returns
A table with two columns: VariableID and VariableResult.

Return type
DataFrame

classmethod **read**(*path*)

Read and initialize variables.

Parameters
path (*Path*) – Path to a two-column .tsv file holding pairs (var_id, var_value). Will use *var_id* to initialize variable, importing dynamically a relevant class from `variables`.

Returns
A dict mapping variable object to its value.

Return type
Variables

write(*path*)

Parameters

- **path** (*Path*) – Path to a file.
- **skip_if_contains** – Skip if a variable ID contains any of the provided strings.

property sequence: *Variables*

Returns
values that are *SequenceVariable* instances.

property structure: *Variables*

Returns
values that are *StructureVariable* instances.

lXtractor.variables.calculator module

Module defining variable calculators managing the exact calculation process of variables on objects.

class lXtractor.variables.calculator.**GenericCalculator**(*num_proc=1*, *valid_exceptions=(<class 'lXtractor.core.exceptions.FailedCalculation'>,)*, *apply_kwargs=None*, *verbose=False*)

Bases: *AbstractCalculator*

Parallel calculator, calculating variables in parallel. Duh.

__call__(*o: OT*, *v: VT*, *m: Mapping[int, int | None] | None*) → tuple[bool, RT]

__call__(*o: Iterable[OT]*, *v: Iterable[VT] | Iterable[Iterable[VT]]*, *m: Iterable[Mapping[int, int | None] | None] | None*) → Iterable[Iterable[tuple[bool, RT]]]

Parameters

- **o** – Object to calculate on.
- **v** – Some variable whose *calculate* method accepts *o*-type instances.
- **m** – Optional mapping between object and some reference object numbering schemes.

Returns
Calculation result.

__init__(*num_proc=1*, *valid_exceptions=(<class 'lXtractor.core.exceptions.FailedCalculation'>,)*, *apply_kwargs=None*, *verbose=False*)

map(*o*, *v*, *m*)

Map variables to a single object.

Parameters

- **o** (*OT*) – Object to calculate on.
- **v** (*Iterable[VT]*) – An iterable over variables whose *calculate* method accepts *o*-type instances.
- **m** (*Mapping[int, int | None] | None*) – Optional mapping between object and some reference object numbering schemes.

Returns
An iterator (generator) over calculation result.

Return type
Generator[tuple[bool, *RT*], None, None]

vmap(*o*, *v*, *m*)

Map objects to a single variable.

Parameters

- **o** (*Iterable[OT]*) – An iterable over objects to calculate on.
- **v** (*VT*) – Some variable whose *calculate* method accepts *o*-type instances.
- **m** (*Iterable[Mapping[int, int | None] | None] | Mapping[int, int | None] | None*) – Optional mapping between object and some reference object numbering schemes.

Returns
An iterator (generator) over calculation result.

Return type
Generator[tuple[bool, *RT*], None, None]

apply_kwargs

num_proc

valid_exceptions

verbose

lXtractor.variables.calculator.**calculate**(*o*, *v*, *m*, *valid_exceptions*, *num_proc*, *verbose=False*, ***kwargs*)

Return type
Generator[*Iterator*[tuple[bool, *RT*]], None, None]

lXtractor.variables.manager module

`Manager` handles variable calculations, such as:

1. Variable manipulations (assignment, deletions, and resetting).
2. **Calculation of variables. Simply manages the calculation process, whereas**
   calculators (*lXtractor.variables.calculator.GenericCalculator* for instance) do the heavy lifting.
3. **Aggregation of the calculation results, either**
   from_chains or *from_iterable*.

class lXtractor.variables.manager.**Manager**(*verbose=False*)

Bases: `object`

Manager of variable calculations, handling assignment, aggregation, and, of course, the calculations themselves.

__init__(*verbose=False*)

Parameters
verbose (*bool*) – Display progress bar.

aggregate_from_chains(*chains*)

Aggregate calculation results from the *variables* container of the provided chains.

```
>>> from lXtractor.variables.sequential import SeqEl
>>> s = lxc.ChainSequence.from_string('abcd', name='_seq')
>>> manager = Manager()
>>> manager.assign([SeqEl(1)], [s])
>>> df = manager.aggregate_from_chains([s])
>>> len(df) == 1
True
>>> list(df.columns)
['VariableID', 'VariableResult', 'ObjectID', 'ObjectType']
```

Parameters
chains (*Iterable[ChainSequence | ChainStructure | tuple[ChainStructure, Ligand]]*) – An iterable over chain sequences/structures.

Returns
A dataframe with *ObjectID*, *ObjectType*, and calculation results.

Return type
DataFrame

aggregate_from_it(*results*, *vs_to_cols=True*, *replace_errors=True*, *replace_errors_with=nan*, *num_vs=None*)

Aggregate calculation results directly from *calculate()* output.

Parameters

- **results** (*Iterable[tuple[ChainSequence | ChainStructure | tuple[ChainStructure, Ligand], SequenceVariable | StructureVariable | LigandVariable, bool, Any]]*) – An iterable over calculation results.

- **vs_to_cols** (*bool*) – If `True`, will attempt to use the wide format for the final results with variables as columns. Otherwise, will use the long format with fixed columns: "ObjectID", "VariableID", "VariableCalculated", and "VariableResult". Note that for the wide format to work, all objects and their variables must have unique IDs.
- **replace_errors** (*bool*) – When calculation failed, replace the calculation results with certain value.
- **replace_errors_with** (*Any*) – Use this value to replace erroneous calculation results.
- **num_vs** (*int | None*) – The number of variables per object. Providing this will significantly increase the aggregation speed.

Returns
A table with results in long or short format.

Return type
DataFrame | dict[str, list]

assign(*vs*, *chains*)

Assign variables to chains sequences/structures.

Parameters

- **vs** (*Sequence[SequenceVariable | StructureVariable | LigandVariable]*) – A sequence of variables.
- **chains** (*Iterable[ChainSequence | ChainStructure | tuple[ChainStructure, Ligand]]*) – An iterable over chain sequences/structures.

Returns
No return. Will store assigned variables within the *variables* attribute.

calculate(*objs*, *vs*, *calculator*, *, *save=False*, ***kwargs*)

Handles variable calculations:

1. Stage calculations (see *stage()*).
2. Calculate variables using the provided calculator.
3. (Optional) save the calculation results to variables container.
4. Output (stream) calculation results.

Note that 3 and 4 are done lazily as calculation results from the calculator become available.

```
>>> from lXtractor.variables.calculator import GenericCalculator
>>> from lXtractor.variables.sequential import SeqEl
>>> s = lxc.ChainSequence.from_string('ABCD', name='_seq')
>>> m = Manager()
>>> c = GenericCalculator()
>>> list(m.calculate([s],[SeqEl(1)],c))
[(_seq|1-4, SeqEl(p=1,_rtype='str',seq_name='seq1'), True, 'A')]
>>> list(m.calculate([s],[SeqEl(5)],c))[0][-2:]
(False, 'Missing index 4 in sequence')
```

Parameters

- **objs** (*Iterable*[ChainSequence | ChainStructure | *tuple*[ChainStructure, Ligand]]) – An iterable over chain sequences/structures.
- **vs** (*Sequence*[SequenceVariable | StructureVariable | LigandVariable] | *None*) – A sequence of variables. If not provided, will use assigned variables (see *assign()*).
- **calculator** (AbstractCalculator) – A calculator object – some callable with the right signature handling the calculations.
- **save** (*bool*) – Save calculation results to variables. Will overwrite any existing matching variables.
- **kwargs** – Passed to *stage()*.

Returns

A generator over tuples: 1. Original object. 2. Variable. 3. Flag indicated whether the calculation was successful. 4. The calculation result (or the error message).

Return type

Generator[tuple[ChainSequence | ChainStructure | tuple[ChainStructure, Ligand], SequenceVariable | StructureVariable | LigandVariable, bool, *Any*], None, None]

remove(*chains*, *vs=None*)

Remove variables from the *variables* container.

Parameters

- **chains** (*Iterable*[ChainSequence | ChainStructure | *tuple*[ChainStructure, Ligand]]) – An iterable over chain sequences/structures.
- **vs** (*Sequence*[SequenceVariable | StructureVariable | LigandVariable] | *None*) – A sequence of variables to remove. If not provided, will remove all variables.

Returns

No return.

reset(*chains*, *vs=None*)

Similar to *remove()*, but instead of deleting, resets variable calculation results.

Parameters

- **chains** (*Iterable*[ChainSequence | ChainStructure | *tuple*[ChainStructure, Ligand]]) – An iterable over chain sequences/structures.
- **vs** (*Sequence*[SequenceVariable | StructureVariable | LigandVariable] | *None*) – A sequence of variables to reset. If not provided, will reset all variables.

Returns

No return.

stage(*chains*, *vs*, ***kwargs*)

Stage objects for calculations (e.g., using *calculate()*). It's a useful method if using a different calculation method and/or parallelization strategy within a *Calculator* class.

See also:

stage() calculate()

```
>>> from lXtractor.variables.sequential import SeqEl
>>> s = lxc.ChainSequence.from_string('ABCD', name='_seq')
>>> m = Manager()
>>> staged = list(m.stage([s], [SeqEl(1)]))
>>> len(staged) == 1
True
>>> staged[0]
(_seq|1-4, 'ABCD', [SeqEl(p=1,_rtype='str',seq_name='seq1')], None)
```

Parameters

- **chains** (*Iterable*[ChainSequence | ChainStructure | *tuple*[ChainStructure, Ligand]]) – An iterable over chain sequences/structures.
- **vs** (*Sequence*[SequenceVariable | StructureVariable | LigandVariable] | *None*) – A sequence of variables. If not provided, will use assigned variables (see *assign()*).
- **kwargs** – Passed to *stage()*.

Returns

An iterable over tuples holding data for variables' calculation.

Return type

Generator[tuple[ChainSequence, *Sequence*[*Any*], *Sequence*[SequenceVariable], *Mapping*[int, int] | None] | tuple[ChainStructure, GenericStructure, *Sequence*[StructureVariable], *Mapping*[int, int] | None], None, None]

verbose

lXtractor.variables.manager.**find_structure**(*s*)

Recursively search for structure up the ancestral tree.

Parameters

s (ChainStructure) – An arbitrary chain structure.

Returns

The first non-empty atom array up the parent chain.

Return type

GenericStructure | None

lXtractor.variables.manager.**get_mapping**(*obj*, *map_name*, *map_to*)

Obtain mapping from a Chain*-type object.

```
>>> s = lxc.ChainSequence.from_string('ABCD', name='_seq')
>>> s.add_seq('some_map', [5, 6, 7, 8])
>>> s.add_seq('another_map', ['D', 'B', 'C', 'A'])
>>> get_mapping(s, 'some_map', None)
{5: 1, 6: 2, 7: 3, 8: 4}
>>> get_mapping(s, 'another_map', 'some_map')
{'D': 5, 'B': 6, 'C': 7, 'A': 8}
```

Parameters

- **obj** (*Any*) – Chain*-type object. If not a Chain*-type object, raises *AttributeError*.
- **map_name** (*str | None*) – The name of a map to create the mapping from. If `None`, the resulting mapping is `None`.
- **map_to** (*str | None*) – The name of a map to create a mapping to. If `None`, will default to the real sequence indices (1-based) for a `ChainSequence` object and to the structure actual numbering for the `ChainStructure`.

Returns

A dictionary mapping from the *map_name* sequence to *map_to* sequence.

Return type

dict | None

`IXtractor.variables.manager.`**stage**(*obj:* ChainStructure, *vs*, ***, *missing*, *seq_name*, *map_name*, *map_to*) → tuple[*ChainStructure*, *GenericStructure*, Sequence[*StructureVariable*], Mapping[int, int] | None]

`IXtractor.variables.manager.`**stage**(*obj:* ChainSequence, *vs*, ***, *missing*, *seq_name*, *map_name*, *map_to*) → tuple[*ChainSequence*, Sequence[Any], Sequence[*SequenceVariable*], Mapping[int, int] | None]

`IXtractor.variables.manager.`**stage**(*obj:* ChainSequence, *vs*, ***, *missing*, *seq_name*, *map_name*, *map_to*) → tuple[*ChainSequence*, Sequence[Any], Sequence[*SequenceVariable*], Mapping[int, int] | None]

`IXtractor.variables.manager.`**stage**(*obj: tuple[*ChainStructure*, *Ligand*]*, *vs*, ***, *missing*, *seq_name*, *map_name*, *map_to*) → tuple[tuple[*ChainStructure*, *Ligand*], *Ligand*, Sequence[*LigandVariable*], Mapping[int, int] | None]

Stage object for calculation. If it's a chain sequence, will stage some sequence/mapping within it. If it's a chain structure, will stage the atom array.

Parameters

- **obj** – A chain sequence or structure or structure-ligand pair to calculate the variables on.
- **vs** – A sequence of variables to calculate.
- **missing** – If `True`, calculate only those assigned variables that are missing.
- **seq_name** – If *obj* is the chain sequence, the sequence name is used to obtain an actual sequence (`obj[seq_name]`).
- **map_name** – The mapping name to obtain the mapping keys. If `None`, the resulting mapping will be `None`.
- **map_to** – The mapping name to obtain the mapping values. See *get_mapping()* for details.

Returns

A tuple with four elements: 1. Original object. 2. Staged target passed to a variable for calculation. 3. A sequence of sequence or structural variables. 4. An optional mapping.

lXtractor.variables.parser module

lXtractor.variables.parser.**init_var**(*var*)

Convert a textual representation of a single variable into a concrete and initialized variable.

```
>>> assert isinstance(init_var('123'), SeqEl)
>>> assert isinstance(init_var('1-2'), Dist)
>>> assert isinstance(init_var('1-2-3-4'), PseudoDihedral)
```

Parameters
var (*str*) – textual representation of a variable.

Returns
initialized variable, a concrete subclass of an `AbstractVariable`

lXtractor.variables.parser.**parse_var**(*inp*)

Parse raw input into a collection of variables, structures, and levels at which they should be calculated.

Parameters
inp (*str*) – `"[variable_specs]--[protein_specs]::[domains]"` format, where:

- ***variable_specs* define the variable type**
  (e.g., *1:CA-2:CA* for CA-CA distance between positions 1 and 2)
- *protein_specs* define proteins for which to calculate variables
- *domains* list the domain names for the given protein collection

Returns
a namedtuple with (1) variables, (2) list of proteins (or `[None]`), and (3) a list of domains (or `[None]`).

lXtractor.variables.sequential module

Module defines variables calculated on sequences

class lXtractor.variables.sequential.**PFP**(*p*, *i*)

Bases: *SequenceVariable*

A ProtFP embedding variable.

See also:

lXtractor.variables.base.ProtFP

__init__(*p*, *i*)

Parameters

- **p** (*int*) – Position, starting from 1.
- **i** (*int*) – A PCA component index starting from 1.

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (*Sequence[str]*) – Some sequence.

- **mapping** (*Mapping[int, int | None] | None*) – Optional mapping between sequence and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
float

i

A PCA component index starting from 1.

p

Position, starting from 1

property rtype: Type[float]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.sequential.**SeqEl**(*p*, *_rtype='str'*, *seq_name='seq1'*)

Bases: *SequenceVariable*[T, T]

A sequence element variable. It doesn't encompass any calculation. Rather, it simply accesses sequence at certain position.

```
>>> v1, v2 = SeqEl(1), SeqEl(1, 'X')
>>> s1, s2 = 'XYZ', [1, 2, 3]
>>> v1.calculate(s1,,
'X'
>>> v2.calculate(s2,,
1
```

__init__(*p*, *_rtype='str'*, *seq_name='seq1'*)

Parameters

- **p** (*int*) – Position, starting from 1.
- **seq_name** (*str*) – The name of the sequence used to distinguish variables pointing to the same position.

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (*Sequence[T]*) – Some sequence.
- **mapping** (*Mapping[int, int | None] | None*) – Optional mapping between sequence and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
T

p

Position, starting from 1.

property rtype: Type[T]

Variable's return type, such that *rtype("result")* converts to the relevant type.

seq_name

Sequence name for which the element is accessed

class lXtractor.variables.sequential.**SliceTransformReduce**(*start=None*, *stop=None*, *step=None*, *seq_name='seq1'*)

Bases: *SequenceVariable*, Generic[T, V, K]

A composite variable with three sequential operations:

1. Slice – subset the sequence (optional).
2. Transform – transform the sequence (optional).
3. Reduce – reduce to a final variable.

This is an abstract class. It requires to define at least two methods:

1. *transform()*.
2. `rtype()` property.

See also:

make_str() – a factory function to quickly make child classes.

__init__(*start=None*, *stop=None*, *step=None*, *seq_name='seq1'*)

Note: *start* and *stop* have inclusive boundaries.

Parameters

- **start** (*int | None*) – Start position
- **stop** (*int | None*) – Stop position.
- **step** (*int | None*) – Slicing step.
- **seq_name** (*str*) – Sequence name. Please use it in case a resulting variable will be applied to seqs other than the primary sequence.

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (*Iterable[K]*) – Some sequence.
- **mapping** (*Mapping[int, int | None] | None*) – Optional mapping between sequence and some reference object numbering schemes.

Returns

A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type

V

abstract static reduce(*seq*)

Reduce the input iterable into the variable result.

Parameters
seq (*Iterable[T] | Iterable[K]*) – Some sort of iterable – the results of the transform (or slicing, if no transformation is used)

Returns
An aggregated value (e.g., float, string, etc.).

Return type
V

static transform(*seq*)

Optionally transform the slicing result. If not used, it is the identity operation.

Parameters
seq (*Iterable[K]*) – The result of slicing operation. If no slicing is used, it is just an `iter(input_seq)`.

Returns
Iterable over transformed elements (can have another type than the input ones).

Return type
Iterable[*T*] | *Iterable*[*K*]

seq_name

Sequence name.

start

Start position.

step

Slicing step.

stop

End position.

lXtractor.variables.sequential.**make_str**(*reduce*, *rtype*, *transform=None*, *reduce_name=None*, *transform_name=None*)

Makes a non-abstract subclass of *SliceTransformReduce* with specific transform and reduce operations.

To make things clearer, transform and reduce operations will have certain names that will be incoroporated into a created class name.

Example 1: no transformation:

```
>>> v_type = make_str(sum, float)
>>> v_type.__name__
'SliceSum'
```

To instanciate it, we provide additional slicing parameters

```
>>> v = v_type(1, 2, seq_name='X')
>>> v.id
"SliceSum(start=1,stop=2,step=None,seq_name='X')"
```

```
>>> v.calculate([1, 2, 3, 4, 5],,
3
```

Example 2: with transformation:

Note that the first operatoiin – slicing – inevitably produces an iterator over the input sequence. Hence, even if we aren't slicing, i.e., provide `None` for all *SliceTransformReduce.__init__()* arguments, we still obtain an iterator over characters. Therefore, we convert it to string and then apply the necessary operation. Note that this feature makes transform `map`-friendly.

```
>>> count_x = lambda x: sum(1 for c in x if c == 'X')
>>> upper = lambda x: "".join(x).upper()
>>> v = make_str(count_x, int, transform=upper, transform_name='upper',
...              reduce_name='countX')()
>>> v.calculate('XoXoxo',,
3
>>> v.id
"SliceUpperCountx(start=None,stop=None,step=None,seq_name='seq1')"
```

See also:

SliceTransformReduce – a base abstract class from which this function generates variables.

Parameters

- **reduce** (*Callable[[Iterable[T]], V]*) – Reduce operation peferably producing a single output.
- **rtype** (*Type*) – Return type of the reduce operation and, since this is the last operatoin, of a variable itself.
- **transform** (*Callable[[Iterator[K]], Iterable[T]] | None*) – Optional transformation operation. It accepts an iterator over (optionally) sliced input elements and returns an iterable over elements of potentially another type, as long as they are supported by the *reduce*.
- **reduce_name** (*str | None*) – The name of the reduce operation. Please provide it in case using `lambda`.
- **transform_name** (*str | None*) – The name of the transform operation. Please provide it in case using `lambda`.

Returns

An uninitialized subclass of *SliceTransformReduce* encapsulating the provided operations within the *SliceTransformReduce.calculate()*.

Return type

Type[SliceTransformReduce]

lXtractor.variables.structural module

Module defining variables calculated on structures.

class lXtractor.variables.structural.**AggDist**(*p1*, *p2*, *key='min'*)

Bases: *StructureVariable*

Aggregated distance between two residues.

It will return `agg_fn(pdist)` where `pdist` is an array of all pairwise distances between atoms of *p1* and *p2*.

__init__(*p1*, *p2*, *key='min'*)

Parameters

- **p1** (*int*) – Position 1.
- **p2** (*int*) – Position 2.
- **key** (*str*) – Agg function name.

Available aggregator functions are:

```
>>> print(list(AggFns))
['min', 'max', 'mean', 'median']
```

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
float

key

Agg function name.

p1

Position 1.

p2

Position 2.

property rtype: Type[float]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.structural.**Chi1**(*p*)

Bases: CompositeDihedral

Chi1-dihedral angle.

static get_dihedrals(*pos*)

Implemented by child classes.

Parameters
pos – Position to create *Dihedral* instances.

Returns
An iterable over *Dihedral*'s. The `calculate()` will try calculating dihedrals in the provided order until the first successful calculation. If no calculations were successful, will raise `FailedCalculation` error.

Return type
list[Dihedral]

class lXtractor.variables.structural.**Chi2**(*p*)

Bases: CompositeDihedral

Chi2-dihedral angle,

static **get_dihedrals**(*pos*)

Implemented by child classes.

Parameters
pos – Position to create *Dihedral* instances.

Returns
An iterable over *Dihedral*'s. The `calculate()` will try calculating dihedrals in the provided order until the first successful calculation. If no calculations were successful, will raise `FailedCalculation` error.

Return type
list[Dihedral]

class lXtractor.variables.structural.**ClosestLigandContactsCount**(*p*, *a=None*)

Bases: *StructureVariable*

The number of atoms involved in contacting ligands.

__init__(*p*, *a=None*)

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
float

a

Atom name. If not provided, sum contacts across all residue atoms.

p

Residue position.

property rtype: Type[int]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.structural.**ClosestLigandDist**(*p*, *a=None*, *agg_lig='min'*, *agg_res='min'*)

Bases: *StructureVariable*

A distance from the selected residue or a residue's atom to a connected ligand.

Each ligand provides *lXtractor.core.ligand.Ligand.dist* array. These arrays are stacked and aggregated atom-wise using *agg_lig*. Then, *agg_res* aggregates the obtained vector of values into a single number.

For instance, to obtain max distance for the closest ligand of a residue 1, use `ClosestLigandDist(1, agg_res='max')`.

If structure has no <ligands `lXtractor.core.structure.GenericStructure.ligands`>, this variable defaults to -1.

..note ::
Attr `lXtractor.core.ligand.dist` provides distances from an atom to the closest ligand atom.

__init__(*p*, *a=None*, *agg_lig='min'*, *agg_res='min'*)

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
float

a

Atom name. If not provided, aggregate across residue atoms.

agg_lig

Aggregator function for ligands.

agg_res

Aggregator function for a residue atoms.

p

Residue position

property rtype: Type[float]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.structural.**ClosestLigandNames**(*p*, *a=None*)

Bases: *StructureVariable*

","-separated contacting ligand (residue) names.

__init__(*p*, *a=None*)

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.

- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
str

a

Atom name. If not provided, merge across all residue atoms.

p

Residue position.

property rtype: Type[str]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.structural.**Contacts**(*p*, *r=5.0*)

Bases: *StructureVariable*

Uses *KDTree* to find atoms within the `r` distance threshold of those defined by target position `p`. Positions these atoms correspond to are returned as a ","-separated string.

If *mapping* is provided, contact positions will be filtered to those covered by this *mapping*.

Note: The default value of `r` is provided by `DefaultConfig["contacts"]["non-covalent"][1]`.

__init__(*p*, *r=5.0*)

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
str

p

Target position.

r

Contact upper bound in angstroms.

property rtype: Type[str]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.structural.**Dihedral**(*p1*, *p2*, *p3*, *p4*, *a1*, *a2*, *a3*, *a4*, *name='GenericDihedral'*)

Bases: *StructureVariable*

Dihedral angle involving four different atoms.

__init__(*p1*, *p2*, *p3*, *p4*, *a1*, *a2*, *a3*, *a4*, *name='GenericDihedral'*)

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
float

a1
Atom name.

a2
Atom name.

a3
Atom name.

a4
Atom name.

property atoms: list[str]

Returns
A list of atoms *a1-a4*.

name: str
Used to designate special kinds of dihedrals.

p1
Position.

p2
Position.

p3
Position.

p4
Position.

property positions: list[int]

Returns
A list of positions *p1-p4*.

property rtype: Type[float]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.structural.**Dist**(*p1*, *p2*, *a1=None*, *a2=None*, *com=False*)

Bases: *StructureVariable*

A distance between two atoms.

__init__(*p1*, *p2*, *a1=None*, *a2=None*, *com=False*)

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns

A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type

float

a1: str | None

Atom name 1.

a2: str | None

Atom name 2.

com: bool

Use center of mass instead of concrete atoms.

p1: int

Position 1.

p2: int

Position 2.

property rtype: Type[float]

Variable's return type, such that *rtype("result")* converts to the relevant type.

class lXtractor.variables.structural.**Omega**(*p*)

Bases: *Dihedral*

Omega dihedral angle.

__init__(*p*)

p

class lXtractor.variables.structural.**Phi**(*p*)

Bases: *Dihedral*

Phi dihedral angle.

__init__(*p*)

p

class lXtractor.variables.structural.**PseudoDihedral**(*p1*, *p2*, *p3*, *p4*)

Bases: *Dihedral*

Pseudo-dihedral angle - "the torsion angle between planes defined by 4 consecutive alpha-carbon atoms."

__init__(*p1*, *p2*, *p3*, *p4*)

class lXtractor.variables.structural.**Psi**(*p*)

Bases: *Dihedral*

Psi dihedral angle.

__init__(*p*)

p

class lXtractor.variables.structural.**SASA**(*p*, *a=None*)

Bases: *StructureVariable*

Solvent-accessible surface area of a residue or a specific atom.

The SASA is calculated for the whole array, and subset to all or a single atoms of a residue (so atoms are taken into account for calculation).

__init__(*p*, *a=None*)

calculate(*obj*, *mapping=None*)

Parameters

- **obj** (GenericStructure) – Some atom array.
- **mapping** (*MappingT | None*) – Optional mapping between structure and some reference object numbering schemes.

Returns
A calculation result of some sensible non-sequence type, such as string, float, int, etc.

Return type
float | None

a

p

property rtype: Type[float]

Variable's return type, such that *rtype("result")* converts to the relevant type.

3.1.6 lXtractor.protocols package

lXtractor.protocols.superpose module

A sandbox module to encapsulate high-level operations based on core *lXtractor*'s functionality.

class lXtractor.protocols.superpose.**SuperposeOutput**(*ID_fix*, *ID_mob*, *RmsdSuperpose*, *Distance*, *Transformation*)

Bases: tuple

Distance: Any

Alias for field number 3

ID_fix: str

Alias for field number 0

ID_mob: str

Alias for field number 1

RmsdSuperpose: float

Alias for field number 2

Transformation: tuple[ndarray, ndarray, ndarray]

Alias for field number 4

lXtractor.protocols.superpose.**align_and_superpose_pair**(*pair*, *dist_fn*, *skip_aln_if_match*)

Use sequence alignment to subset each chain structure in *pair* to common aligned residues and common atoms in each aligned residue pair. Use *superpose_pair()* to superpose the atom arrays from subsetted chain structures.

Parameters

- **pair** (*tuple[tuple[str, ChainStructure, AtomArray | None], tuple[str, ChainStructure, AtomArray | None]]*) – A pair of staged inputs.
- **dist_fn** (*Callable[[AtomArray, AtomArray], Any] | None*) – An optional distance function accepting two positional args: "fixed" atom array and superposed atom array.
- **skip_aln_if_match** (*str*) – Passed to lXtractor.core.chain.subset_to_matching().

Returns

a tuple with id_fixed, id_mobile, rmsd of the superposed atoms, calculated distance, and the transformation matrices.

Return type

tuple[str, str, float, *Any*, tuple[*ndarray*, *ndarray*, *ndarray*]]

lXtractor.protocols.superpose.**superpose_pair**(*pair*, *dist_fn*)

A function performing superposition and rmsd calculation of already prepared `AtomArray` objects. Each must have the same number of atoms.

Parameters

- **pair** (*tuple[tuple[str, AtomArray, AtomArray | None], tuple[str, AtomArray, AtomArray | None]]*) – A pair of staged inputs. A staged input is a tuple with an identifier, an atom array to superpose, and an optional atom array for the *dist_fn*.
- **dist_fn** (*Callable[[AtomArray, AtomArray], Any] | None*) – An optional distance function accepting two positional args: "fixed" atom array and superposed atom array.

Returns

a tuple with id_fixed, id_mobile, rmsd of the superposed atoms, calculated distance, and the transformation matrices.

Return type

tuple[str, str, float, *Any*, tuple[*ndarray*, *ndarray*, *ndarray*]]

lXtractor.protocols.superpose.**superpose_pairwise**(*fixed*, *mobile=None*, *selection_superpose=(None, None)*, *selection_dist=None*, *dist_fn=None*, ***, *strict=True*, *map_name=None*, *exclude_hydrogen=False*, *skip_aln_if_match='len'*, *verbose=False*, *num_proc=1*, ***kwargs*)

Superpose pairs of structures. Two modes are available:

1. `strict=True` – potentially faster and memory efficient, more parallelization friendly. In this case, after selection using the provided positions and atoms, the number of atoms between each fixed and mobile structure must match exactly.

2. `strict=False` – a "flexible" protocol. In this case, after the selection of atoms, there are two additional steps:

    1. Sequence alignment between the selected subsets. It's guaranteed to produce the same number of residues between fixed and mobile, which may be less than the initially selected number (see `subset_to_matching()`).

    2. Following this, subset each pair of residues between fixed and mobile to a common list of atoms (see *filter_to_common_atoms*).

As a result, the "flexible" mode may be suitable for distantly related structures, while the "strict" mode may be used whenever it's guaranteed that the selection will produce the same sets of atoms between fixed and mobile.

See also:

`lXtractor.util.structure.filter_selection_extended()` – used to apply the selections.

Parameters

- **fixed** (*Iterable[ChainStructure]*) – An iterable over chain structures that won't be moved.
- **mobile** (*Iterable[ChainStructure] | None*) – An iterable over chain structures to superpose onto fixed ones. If `None`, will use the combinations of *fixed*.
- **selection_superpose** (*tuple[Sequence[int] | None, Sequence[Sequence[str]] | Sequence[str] | None] | Callable[[ChainStructure], AtomArray]*) – A tuple with (residue positions,

atom names) to select atoms for superposition, which will be applied to each *fixed* and *mobile* structure. If `(None, None)`, will use all positions and atoms. Alternatively, a selector function accepting a chain structure and returning an atom array. If *strict* is `False`, it will convert the selected atom array to a chain structure.

- **selection_dist** (*tuple[Sequence[int] | None, Sequence[Sequence[str]] | Sequence[str] | None] | Callable[[ChainStructure], AtomArray] | None*) – Same as *selection_superpose*. In addition, accepts `None` to indicate an empty selection, in which case, *dist_fn* should also be `None`.
- **dist_fn** (*Callable[[AtomArray, AtomArray], Any] | None*) – An optional distance function applied to a pair of superposed atom arrays, possibly different from the arrays selected for superposition, which is controlled via *selection_dist*.
- **map_name** (*str | None*) – Mapping for positions in both selection arguments. If used, must exist within `Seq` of each fixed and mobile structure. A good candidate is a mapping to a reference sequence or *Alignment*.
- **exclude_hydrogen** (*bool*) – Exclude all hydrogen atoms during selection.
- **strict** (*bool*) – Enable/disable the "strict" protocol. See the explanation above.
- **skip_aln_if_match** (*str*) – Skip the sequence alignment if this field matches.
- **verbose** (*bool*) – Display progress bar.
- **num_proc** (*int*) – The number of parallel processes. For large selections, may consume a lot of RAM, so caution advised.
- **kwargs** – Passed to `ProcessPoolExecutor.map()`. Useful for controlling *chunksize* and *timeout* parameters.

Returns

A generator of `namedtuple` outputs each containing the IDs of the superposed objects, the RMSD between superposed structures, the distance function output, and the transformation matrices.

Return type

Generator[SuperposeOutput, None, None]

3.1.7 IXtractor.collection package

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Gerard JP van Westen, Remco F Swier, Jörg K Wegner, Adriaan P IJzerman, Herman WT van Vlijmen, and Andreas Bender. Benchmarking of protein descriptor sets in proteochemometric modeling (part 1): comparative study of 13 amino acid descriptor sets. *Journal of Cheminformatics*, 5(1):41–41, 2013. doi:10.1186/1758-2946-5-41.

[2] Gerard JP van Westen, Remco F Swier, Isidro Cortes-Ciriano, Jörg K Wegner, John P Overington, Adriaan P IJzerman, Herman WT van Vlijmen, and Andreas Bender. Benchmarking of protein descriptor sets in proteochemometric modeling (part 2): modeling performance of 13 amino acid descriptor sets. *Journal of Cheminformatics*, 5(1):42, 2013. doi:10.1186/1758-2946-5-42.

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W