stam-python

Release version = "0.4.0"

Maarten van Gompel

Jan 24, 2024

CONTENTS

STAM is a data model for stand-off text annotation and described in detail here. This is a python library (to be more specific; a python binding written in Rust) to work with the model.

What can you do with this library?

- Keep, build and manipulate an efficient in-memory store of texts and annotations on texts
- **Search in annotations, data and text, either programmatically or via the STAM Query Language.**
  - Search annotations by data, textual content, relations between text fragments (overlap, embedding, adjacency, etc),
  - Search in text (incl. via regular expressions) and find annotations targeting found text selections.
  - Search in data (set,key,value) and find annotations that use the data.
  - Elementary text operations with regard for text offsets (splitting text on a delimiter, stripping text).
  - Convert between different kind of offsets (absolute, relative to other structures, UTF-8 bytes vs unicode codepoints, etc)
- **Read and write resources and annotations from/to STAM JSON, STAM CSV, or an optimised binary (CBOR) representation**
  - The underlying STAM modelaims to be clear and simple. It is flexible and does not commit to any vocabulary or annotation paradigm other than stand-off annotation.

This STAM library is intended as a foundation upon which further applications can be built that deal with stand-off annotations on text. We implement all the low-level logic in dealing this so you no longer have to and can focus on your actual application.

This library offers a higher-level interface than the underlying Rust library. We aim to implement the full model and most extensions.

CHAPTER

ONE

TUTORIAL

A tutorial for working with this API is available in the form of an interactive Jupyter Notebook: STAM Tutorial: Standoff Text Annotation for Pythonistas.

CHAPTER

TWO

CONTENTS

2.1 API Reference

This page contains auto-generated API reference documentation[1].

[1] Created with sphinx-autoapi

2.1.1 stam

2.1.1.1 Module Contents

2.1.1.1.1 Classes

Annotation	*Annotation* represents a particular *instance of annotation* and is the central
AnnotationData	AnnotationData holds the actual content of an annotation; a key/value pair. (the
AnnotationDataSet	An *AnnotationDataSet* stores the keys (*DataKey*) and values
AnnotationStore	An Annotation Store is a collection of annotations, resources and
Annotations	An *Annotations* object holds an arbitrary collection of annotations.
Cursor	A cursor points to a specific point in a text. It is used to select offsets. Units are unicode codepoints (not bytes!)
Data	A *Data* object holds an arbitrary collection of annotation data.
DataKey	The DataKey class defines a vocabulary field, it
DataValue	Encapsulates a value and its type. Held by *AnnotationData*. This type is not a reference but holds the actual value.
Offset	Text selection offset. Specifies begin and end offsets to select a range of a text, via two *Cursor* instances.
Selector	A *Selector* identifies the target of an annotation and the part of the
SelectorKind	An enumeration of possible selector types
TextResource	This holds the textual resource to be annotated. It holds the full text in memory.
TextSelection	This holds a slice of a text.
TextSelectionOperator	The TextSelectionOperator, simply put, allows comparison of two *TextSelection* instances. It
TextSelections	A *TextSelections* object holds an arbitrary collection of text selections.

class stam.**Annotation**

Annotation represents a particular *instance of annotation* and is the central concept of the model. Annotations can be considered the primary nodes of the graph model. The instance of annotation is strictly decoupled from the *data* or key/value of the annotation (*AnnotationData*). After all, multiple instances can be annotated with the same label (multiple annotations may share the same annotation data). Moreover, an *Annotation* can have multiple annotation data associated. The result is that multiple annotations with the exact same content require less storage space, and searching and indexing is facilitated.

This structure is not instantiated directly, only returned. Use *AnnotationStore.annotate()* to instantiate a new Annotation.

__iter__() → Iterator[*AnnotationData*]

Returns a iterator over all data (*AnnotationData*) in this annotation; this has little overhead but is less suitable if you want to do further filtering, use *data()* instead for that.

Return type
Iterator[*AnnotationData*]

__len__() → int

Returns the number of data items (*AnnotationData*) in this annotation

Return type
int

__str__() → str

Returns the text of the annotation. If the annotation references multiple text slices, they will be concatenated with a space as a delimiter, but note that in reality the different parts may be non-contingent!

Use *text()* instead to retrieve a list of texts

Return type
str

annotations(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation* instances) that are referring to this annotation (i.e. others using an AnnotationSelector).

The annotations can be filtered using positional and/or keyword arguments.

Parameters

- ***args** (*tuple, optional*) – These arguments can any be of the following types:
  - *DataKey*
    Returns annotations with data matching this key.
  - *AnnotationData*
    Returns only annotations that have this exact data.
  - *Annotations* | *Annotation*
    Returns only annotations that match any of those specified here.
  - *Data* | *AnnotationData*
    Returns only annotations with data matching any of those specified here.
  - dict **with keys:**
    - **set** - An ID of a dataset (or a DataAnnotationSet instance), only needed when specifying *key* as a string (see below)
    - **key** - A key, either an instance of *DataKey* or a string, in the latter case you need to specify *set* as well.
    - **value** - (see keyword arguments below)
- ****kwargs** (*dict, optional*) –
  - **limit: (Optional[int] = None)**
    The maximum number of results to return (default: unlimited)
  - **set: (Optional[Union[str,AnnotationDataSet]] = None)**
    An ID of a dataset (or an *AnnotationDataSet* instance), only needed when specifying *key* as a string
  - **key: (Optional[Union[str,DataKey]] = None)**
    An ID of a key (or a *DataKey* instance), make sure to specify *set* as well if you use a string value for this parameter.

- **value: (Optional[Union[str,int,float,bool]])**
  Constrain the search to annotations with data of a certain value. This can only be used when you also pass a *DataKey* as filter. This holds the exact value to search for, there are other variants of this keyword available, see *data()* for a full list.
- **limit: (Optional[int] = None)**
  The maximum number of results to return (default: unlimited)

Return type
Annotations

Example

Filter by data key and value:

```
key = store.dataset("linguistic-set").key("part-of-speech")
for annotation in store.annotations(key, value="noun"):
    ...
```

But if you already have the key, like in the example above, you may just as well do (more efficient):

```
for annotation in key.annotations(value="noun"):
    ...
```

annotations_in_targets(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation* instances) this annotation refers to (i.e. using an *AnnotationSelector*)

The annotations can be filtered using positional and/or keyword arguments; see *annotations()* for full documentation. One extra keyword argument is available for this method (see below).

Annotations will returned be in textual order unless recursive is set or a DirectionalSelector is involved.

Keyword Arguments
recursive (*bool*) – Follow AnnotationSelectors recursively (default False)

Return type
Annotations

data(**args*, ***kwargs*) → *Data*

Returns annotation data (*Data* containing *AnnotationData*) used by this annotation.

The data can be filtered using keyword arguments. If you don't care for any filtering and just want a simple iterator overlap the data, then just iterating over the annotation directly (*__iter__()*) will be more efficient. Do note that implementing any filtering yourself in Python is much less performant than letting this data method do it for you.

Parameters

- ***args** (*tuple, optional*) – Filter arguments, these can be of the following types:
  - *DataKey*
    Returns data matching this key
  - *Annotation*
    Returns data referenced by the mentioned annotation
  - *AnnotationData*
    Returns only this exact data. Not very useful, use *test_data()* instead.

  - *Annotations* | [class:*Annotation*]
    Returns data references by annotations in the provided collection.
  - *Data* | [class:*AnnotationData*]
    Returns only data that is in the provided *Data* collection (intersection)
  - dict with keys:
    * **set** - An ID of a dataset (or a DataAnnotationSet instance), only needed when specifying *key* as a string (see below)
    * **key** - A key, either an instance of *DataKey* or a string, in the latter case you need to specify *set* as well.
    * **value** or variants (see keyword arguments below)

- ****kwargs** (*dict, optional*) –
  - **limit:** ***Optional[int] = None***
    The maximum number of results to return (default: unlimited)
  - **set:** ***Optional[Union[str,AnnotationDataSet]] = None***
    An ID of a dataset (or an *AnnotationDataSet* instance), only needed when specifying *key* as a string
  - **key:** ***Optional[Union[str,DataKey]] = None***
    An ID of a key (or a *DataKey* instance), make sure to specify *set* as well if you use a string value for this parameter.
  - **value:** ***Optional[Union[str,int,float,bool,List[Union[str,int,float,bool]]]]***
    Search for data matching a specific value. This holds exact value to search for. Further variants of this keyword are listed below:
  - **value_not:** ***Optional[Union[str,int,float,bool]]***
    Value must not match
  - **value_greater:** ***Optional[Union[int,float]]***
    Value must be greater than specified (int or float)
  - **value_less:** ***Optional[Union[int,float]]***
    Value must be less than specified (int or float)
  - **value_greatereq:** ***Optional[Union[int,float]]***
    Value must be greater than specified or equal (int or float)
  - **value_lesseq:** ***Optional[Union[int,float]]***
    Value must be less than specified or equal (int or float)
  - **value_in:** ***Optional[Tuple[Union[str,int,float,bool]]]***
    Value must match any in the tuple (this is a logical OR statement)
  - **value_not_in:** ***Optional[Tuple[Union[str,int,float,bool]]]***
    Value must not match any in the tuple
  - **value_in_range:** ***Optional[Tuple[Union[int,float]]]***
    Must be a numeric 2-tuple with min and max (inclusive) values
  - **limit:** ***Optional[int] = None***
    The maximum number of results to return (default: unlimited)

Return type
Data

Example

Get all part-of-speech data pertaining to this annotation:

```
key = store.dataset("linguistic-set").key("part-of-speech")
for data in annotation.data(filter=key):
    ...
```

datasets(*limit: int | None = None*) → List[*AnnotationDataSet*]

Returns a list of annotation data sets (*AnnotationDataSet*) this annotation refers to. This only returns the ones referred to via a *DataSetSelector*, i.e. as metadata.

Parameters
limit (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)

Return type
List[*AnnotationDataSet*]

has_id(*id: str*) → bool

Tests the ID

Parameters
id (*str*) –

Return type
bool

id() → str | None

Returns the public ID (by value, aka a copy) Don't use this for extensive ID comparisons, use *has_id()* instead as it is more performant (no copy).

Return type
Optional[str]

offset() → *Offset* | None

Returns the offset this annotation's selector targets, exactly as specified

Return type
Optional[*Offset*]

related_text(*operator:* TextSelectionOperator, **args*, ***kwargs*) → *TextSelections*

Applies a *TextSelectionOperator* to find all other text selections who are in a specific relation with the ones from the current annotation. Returns a collection *TextSelections* containing all matching *TextSelection* instances.

Text selections will be returned in textual order. They may be filtered via positional and/or keyword arguments. See *Annotation.textselections()*.

If you are interested in the annotations associated with the found text selections, then add *.annotations()* to the result.

Parameters
operator (*TextSelectionOperator*) – The operator to apply when comparing text selections

Keyword Arguments
limit (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)

Return type
TextSelections

See *Annotation.textselections()* for further keyword arguments to filter.

Examples

Find all text selections that overlap with the annotation:

```
for textselection in annotation.related_text(TextSelectionOperator.overlaps()):
    ...
```

If you want to get the annotations instead, just add `.annotations()`:

```
for annotations in annotation.related_text(TextSelectionOperator.overlaps()).
↪annotations():
    ...
```

Assume *sentence* is an annotation representing a sentence, we can find text selections inside (embedded in) the sentence as follows:

```
for textselection in sentence.related_text(TextSelectionOperator.embeds()):
    ...
```

Like above, but now we actively look for annotations that are marked as words, effectively selecting all words in a sentence:

```
data_word = store.dataset("structural-set").key("type").data(value="word",␣
↪limit=1)[0]
for word in sentence.related_text(TextSelectionOperator.embeds()).
↪annotations(filter=data_word):
    ...
```

resources(*limit: int | None = None*) → List[*TextResource*]

Returns a list of resources (*TextResource*) this annotation refers to

Parameters
limit (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)

Return type
List[*TextResource*]

select() → *Selector*

Returns a selector pointing to this annotation

Return type
Selector

selector_kind() → *SelectorKind*

Returns the type of the selector of this annotation

Return type
SelectorKind

target() → *Selector*

Returns the target selector (*Selector*) for this annotation. This is mainly useful if you want to add another annotation pointing to the same target.

Return type
Selector

test_annotations(**args*, ***kwargs*) → bool

Tests whwther there are annotations (*Annotations* containing *Annotation*) that are referring to this annotation (i.e. others using an AnnotationSelector). This method is like *annotations()*, but only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using keyword arguments. See *Annotation.annotations()*.

Example

Filter by data key and value:

```
key = store.dataset("linguistic-set").key("part-of-speech")
for annotation in store.annotations_in_targets(filter=key, value="noun"):
    ...
```

Return type
bool

test_data(**args*, ***kwargs*) → bool

Tests whether certain annotation data is used by this annotation. The data can be filtered using positional and/or keyword arguments. See *data()*. Unlike *data()*, this method merely tests without returning the data, and as such is more performant.

Return type
bool

text() → List[str]

Returns the text of the annotation. Note that this will always return a list (even it if only contains a single element), as an annotation may reference multiple texts.

If you are sure an annotation only reference a single contingent text slice or are okay with slices being concatenated, then you can use the *str()* function instead.

Return type
List[str]

textselections(***kwargs*) → *TextSelections*

Returns a collection of all textselections (*TextSelection*) referenced by the annotation (i.e. via a *TextSelector*). Note that this will always return a collection (even it if only contains a single element), as an annotation may reference multiple text selections.

Text selections will be returned in textual order, except if a DirectionalSelector was used.

Text selections may be filtered using the following positionl and/or keyword arguments:

Parameters

- ***args** (*tuple, optional*) – Filter arguments, can be of the following types:
  - *DataKey*
    Returns text selections referenced by annotations with data matching this key

- *AnnotationData*
  Returns text selections referenced by annotations that have this exact data
- *Annotations* | [*Annotation*]
  Returns text selections referenced by any annotations that are already in the provided *Annotations* collection (intersection)
- *Data* | [*AnnotationData*]
  Returns only textselections referenced by annotations with data that is in the provided collection.
- `dict` **with keys:**
  - **set** - An ID of a dataset (or a `DataAnnotationSet` instance), only needed when specifying *key* as a string (see below)
  - **key** - A key, either an instance of *DataKey* or a string, in the latter case you need to specify *set* as well.
  - **value** (see keyword arguments below)

- **`**kwargs`** (*dict, optional*) –

  limit: Optional[int] = None
  The maximum number of results to return (default: unlimited)

  value: Optional[Union[str,int,float,bool]]
  Constrain the search to text selections referenced by annotations with data of a certain value. This is usually used together with passing a *DataKey* as filter in the positional arguments. This holds the exact value to search for, there are other variants of this keyword available, see *data()* for a full list.

Return type
TextSelections

class `stam.`**AnnotationData**

AnnotationData holds the actual content of an annotation; a key/value pair. (the term *feature* is regularly seen for this in certain annotation paradigms). Annotation Data is deliberately decoupled from the actual *Annotation* instances so multiple annotation instances can point to the same content without causing any overhead in storage. Moreover, it facilitates indexing and searching. The annotation data is part of an *AnnotationDataSet*, which effectively defines a certain user-defined vocabulary.

Once instantiated, instances of this type are, by design, largely immutable. The key and value can not be changed. Create a new AnnotationData and new Annotation for edits. This class is not instantiated directly.

annotations(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation*) that make use of this data.

The annotations can be filtered using positional and/or keyword arguments.

Parameters

- **`*args`** (*tuple, optional*) – Filter arguments, can any be of the following types:
  - *DataKey*
    Returns annotations with data matching this key.
  - *AnnotationData*
    Returns only annotations that have this exact data.
  - *Annotations* | *Annotation*
    Returns only annotations that match any of those specified here.

- *Data* | *AnnotationData*
  Returns only annotations with data matching any of those specified here.
- `dict` **with keys:**
  - **set** - An ID of a dataset (or a `DataAnnotationSet` instance), only needed when specifying *key* as a string (see below)
  - **key** - A key, either an instance of *DataKey* or a string, in the latter case you need to specify *set* as well.
  - **value** - (see keyword arguments below)

- **`**kwargs`** (*dict, optional*) –
  - **limit: (Optional[int] = None)**
    The maximum number of results to return (default: unlimited)
  - **set: (Optional[Union[str,AnnotationDataSet]] = None)**
    An ID of a dataset (or an *AnnotationDataSet* instance), only needed when specifying *key* as a string
  - **key: (Optional[Union[str,DataKey]] = None)**
    An ID of a key (or a *DataKey* instance), make sure to specify *set* as well if you use a string value for this parameter.
  - **value: (Optional[Union[str,int,float,bool]])**
    Constrain the search to annotations with data of a certain value. This can only be used when you also pass a *DataKey* as filter. This holds the exact value to search for, there are other variants of this keyword available, see `data()` for a full list.
  - **limit: (Optional[int] = None)**
    The maximum number of results to return (default: unlimited)

Return type
Annotations

`annotations_len`(*limit: int | None = None*) → int

Returns the number of annotations (*Annotation*) that use this data. Note that this is much faster than doing *len(annotations())*!

Parameters
`limit` (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)

Return type
int

`dataset`() → *AnnotationDataSet*

Returns the *AnnotationDataSet* this data is part of

Return type
AnnotationDataSet

`has_id`(*id: str*) → bool

Tests the ID

Parameters
`id` (*str*) –

Return type
bool

id() → str | None

Returns the public ID (by value, aka a copy) Don't use this for extensive ID comparisons, use *has_id()* instead as it is more performant (no copy).

Return type
Optional[str]

key() → *DataKey*

Basic retrieval method to obtain the key

Return type
DataKey

select() → *Selector*

Returns a selector pointing to this data (AnnotationDataSelector)

Return type
Selector

test_annotations(**args*, ***kwargs*) → bool

Tests whether there are any annotations that make use of this data. This method is like *annotations()*, but only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using keyword arguments. See *Annotation.annotations()*.

Return type
bool

test_value(*reference:* DataValue) → bool

Tests whether the value equals another This is more efficient than calling *value()*] and doing the comparison yourself.

Parameters
reference (DataValue) –

Return type
bool

value() → *DataValue*

Basic retrieval method to obtain the value

Return type
DataValue

class stam.**AnnotationDataSet**

An *AnnotationDataSet* stores the keys (*DataKey*) and values *AnnotationData* (which in turn encapsulates *DataValue*) that are used by annotations.

It effectively defines a certain vocabulary, i.e. key/value pairs. The *AnnotationDataSet* does not store the *Annotation* instances, those are in the *AnnotationStore*. The datasets themselves are also held by the *AnnotationStore*.

Use AnnotationStore.add_annotationset() to instantiate a new AnnotationDataSet, it can not be constructed directly.

__iter__() → Iterator[*AnnotationData*]

Returns an iterator over all *AnnotationData* in the dataset. If you want to do any filtering, use *data()* instead.

Return type
Iterator[*AnnotationData*]

add_data(*key: str*, *value:* DataValue | *str* | *float* | *int* | *list* | *bool*, *id: str* | *None* = *None*) → *AnnotationData*

Create a new *AnnotationData* instances and add it to the dataset. Returns the added data.

Parameters

- **key** (*str*) –
- **value** (*Union[DataValue, str, float, int, list, bool]*) –
- **id** (*Optional[str]*) –

Return type
AnnotationData

add_key(*key: str*) → *DataKey*

Create a new *DataKey* and adds it to the dataset. Returns the added key.

Parameters
key (*str*) –

Return type
DataKey

annotationdata(*id: str*) → *AnnotationData*

Basic retrieval method to obtain annotationdata from a dataset, by ID

Parameters
id (*str*) –

Return type
AnnotationData

data(**args*, ***kwargs*) → *Data*

Returns annotation data (*Data* containing *AnnotationData*) used by this key.

The data can be filtered using positional and/or keyword arguments. See *Annotation.data()*. If you don't intend to do any filtering at all, then just using *__iter__()* may be faster.

Return type
Data

data_len() → int

Returns the number of annotation data instances in the set

Return type
int

has_id(*id: str*) → bool

Tests the ID

Parameters
id (*str*) –

Return type
bool

id() → str | None

Returns the public ID (by value, aka a copy) Don't use this for extensive ID comparisons, use *has_id()* instead as it is more performant (no copy).

Return type
Optional[str]

key(*key: str*) → *DataKey*

Basic retrieval method to obtain a key from a dataset

Parameters
key (*str*) –

Return type
DataKey

keys() → Iterator[*DataKey*]

Returns an iterator over all *DataKey* instances in the dataset

Return type
Iterator[*DataKey*]

keys_len() → int

Returns the number of keys in the set

Return type
int

select() → *Selector*

Returns a selector pointing to this annotation dataset (via a *DataSetSelector*)

Return type
Selector

test_data(**args*, ***kwargs*) → bool

Tests whether certain annotation data exists in this set. The data can be filtered using positional and/or keyword arguments. See *Annotation.data()*. This method is like *data()*, but merely tests without returning the data, and as such is more performant.

Return type
bool

class stam.**AnnotationStore**(*id=None*, *file=None*, *string=None*, *config=None*)

An Annotation Store is a collection of annotations, resources and annotation data sets. It can be seen as the *root* of the *graph model* and the glue that holds everything together. It is the entry point for any stam model.

To instantiate an AnnotationStore, at least one of *id*, *file* or *string* must be specified as keyword arguments:

Keyword Arguments

- **id** (*Optional[str], default: None*) – The public ID for a *new* store
- **file** (*Optional[str], default: None*) – The STAM JSON, STAM CSV or STAM CBOR file to load
- **string** (*Optional[str], default: None*) – STAM JSON as a string
- **config** (*Optional[dict]*) – A python dictionary containing configuration parameters:
  - **use_include: Optional[bool], default: True**
    Use the @*include* mechanism to point to external files, if unset, all data will be kept in a single STAM JSON file.
  - **debug: Optional[bool], default: False**
    Enable debug mode, outputs extra information to standard error output (verbose!)
  - **annotation_annotation_map: Optional[bool], default: True**
    Enable/disable index for annotations that reference other annotations

- **resource_annotation_map: Optional[bool], default: True**
  Enable/disable reverse index for TextResource => Annotation. Holds only annotations that **directly** reference the TextResource (via a ResourceSelector), i.e. metadata
- **dataset_annotation_map: Optional[bool], default: True**
  Enable/disable reverse index for AnnotationDataSet => Annotation. Holds only annotations that **directly** reference the AnnotationDataSet (via DataSetSelector), i.e. metadata
- **key_annotation_metamap: Optional[bool], default: True**
  Enable/disable reverse index for DataKey => Annotation. Holds only annotations that **directly** reference the DataKey (via DataKeySelector), i.e. metadata
- **data_annotation_metamap: Optional[bool], default: True**
  Enable/disable reverse index for AnnotationData => Annotation. Holds only annotations that **directly** reference the AnnotationData (via AnnotationDataSelector), i.e. metadata
- **textrelationmap: Optional[bool], default: True**
  Enable/disable the reverse index for text, it maps TextResource => TextSelection => Annotation
- **generate_ids: Optional[bool], default: False**
  Generate pseudo-random public identifiers when missing (during deserialisation). Each will consist of 21 URL-friendly ASCII symbols after a prefix of A for Annotations, S for DataSets, D for AnnotationData, R for resources
- **strip_temp_ids: Optional[bool], default: True**
  Strip temporary IDs during deserialisation. Temporary IDs start with an exclamation mark, a capital ASCII letter denoting the type, and a number
- **shrink_to_fit: Optional[bool], default: True**
  Shrink data structures to optimize memory (at the cost of longer deserialisation times)
- **milestone_interval: Optional[int], default: 100**
  Milestone placement interval (in unicode codepoints) in indexing text resources. A low number above zero increases search performance at the cost of memory and increased initialisation time.

Example

Load a store from file:

```
store = AnnotationStore(file="hamlet.store.json")
```

Instantiate a store from scratch and populate it with a resource and annotation:

```
self.store = AnnotationStore(id="test")
resource = self.store.add_resource(id="testres", text="Hello world")
self.store.annotate(id="A1",
                    target=Selector.textselector(resource, Offset.simple(6,11)),
                    data={ "id": "D1", "key": "pos", "value": "noun", "set":
↪"testdataset"})
```

__iter__() → Iterator[*Annotation*]

Returns an iterator over all annotations (*Annotation*) in this store.

This iterator has little runtime overhead but does not provide any filtering options, use *annotations()* instead if you plan to do any filtering, or use the equally named method on other objects for more constrained and filterable annotations (e.g. *DataKey.annotations()*, AnnotationDataSet.annotations(), *TextResource.annotations()*)

Return type
Iterator[*Annotation*]

add_dataset(*id: str*) → *AnnotationDataSet*

Create a new *AnnotationDataSet* and add it to the store. Returns the added instance.

Parameters
id (*str*) –

Return type
AnnotationDataSet

add_resource(*filename: str | None = None*, *text: str | None = None*, *id: str | None = None*) → *TextResource*

Create a new *TextResource* and add it to the store. Returns the added instance.

Parameters

- **filename** (*Optional[str]*) –
- **text** (*Optional[str]*) –
- **id** (*Optional[str]*) –

Return type
TextResource

annotate(*target:* Selector, *data: dict | List[dict] |* AnnotationData *| List[*AnnotationData*]*, *id: str | None = None*) → *Annotation*

Adds a new annotation. Returns the *Annotation* instance that was just created.

Parameters

- **target** (*Selector*) – A target selector that determines the object of annotation
- **data** (*Union[dict,List[dict],*AnnotationData*, List[*AnnotationData*]]*) – A dictionary or list of dictionaries with data to set. The dictionary may have fields: *id* (optional),`key`,`set`, and *value*. Alternatively, you can pass an existing *AnnotationData* instance.
- **id** (*Optional[str]*) – The public ID for the annotation. If unset, one may be autogenerated if this was explicitly enabled in the configuraiton.

Return type
Annotation

Example

Instantiate a store from scratch and populate it with a resource and annotation:

```
self.store.annotate(id="A1",
                    target=Selector.textselector(store.resource("testres"),␣
↪Offset.simple(6,11)),
                    data={ "id": "D1", "key": "pos", "value": "noun", "set":
↪"testdataset"})
```

annotation(*id: str*) → *Annotation*

Basic retrieval method that returns an *Annotation* by ID. Raises an exception if not found.

Parameters
id (*str*) –

Return type
Annotation

annotationdata(*set_id: str*, *data_id: str*) → *AnnotationData*

Shortcut retrieval method that returns an *AnnotationData* by ID

Parameters

- **set_id** (*str*) –
- **data_id** (*str*) –

Return type
AnnotationData

annotations(**args*, ***kwargs*) → *Annotations*

Returns an iterator over all annotations (*Annotation*) in this store.

Filtering can be applied using positional arguments and/or keyword arguments. It is recommended to only use this method if you apply further filtering, otherwise the memory overhead may be very large if you have many annotations. Otherwise you can fall back to a more low-level iterator, *__iter__()* instead

Parameters

- ***args** (*tuple, optional*) – Filter arguments. These can any be of the following types:
  - *DataKey*
    Returns annotations with data matching this key.
  - *AnnotationData*
    Returns only annotations that have this exact data.
  - *Annotations* | [*Annotation*]
    Returns only annotations that match any of those specified here.
  - *Data* | [*AnnotationData*]
    Returns only annotations with data matching any of those specified here.
  - **dict with keys:**
    * **set** - An ID of a dataset (or a `DataAnnotationSet` instance), only needed when specifying *key* as a string (see below)
    * **key** - A key, either an instance of *DataKey* or a string, in the latter case you need to specify *set* as well.

        * **value** - (see keyword arguments below)

    - ****kwargs** (*dict, optional*) –
        - **limit: (Optional[int] = None)**
          The maximum number of results to return (default: unlimited)
        - **set: (Optional[Union[str,AnnotationDataSet]] = None)**
          An ID of a dataset (or an *AnnotationDataSet* instance), only needed when specifying *key* as a string
        - **key: (Optional[Union[str,DataKey]] = None)**
          An ID of a key (or a *DataKey* instance), make sure to specify *set* as well if you use a string value for this parameter.
        - **value: (Optional[Union[str,int,float,bool]])**
          Constrain the search to annotations with data of a certain value. This can only be used when you also pass a *DataKey* as filter. This holds the exact value to search for, there are other variants of this keyword available, see *data()* for a full list.

Return type
Annotations

annotations_len() → int

Returns the number of annotations in the store (not substracting deletions)

Return type
int

data(**args*, ***kwargs*) → *Data*

Returns an iterator over all data (*AnnotationData*) in this store.

Filtering can be applied using positional arguments and/or keyword arguments. It is recommended to only use this method if you apply further filtering, otherwise the memory overhead may be very large if you have a lot of data.

Parameters

- ***args** (*tuple, optional*) – Filter arguments, these can be of the following types:
    - ***DataKey***
      Returns data matching this key
    - ***Annotation***
      Returns data referenced by the mentioned annotation
    - ***AnnotationData***
      Returns only this exact data. Not very useful, use `test_data()` instead.
    - ***Annotations* | [class:*Annotation*]**
      Returns data references by annotations in the provided collection.
    - ***Data* | [class:*AnnotationData*]**
      Returns only data that is in the provided *Data* collection (intersection)
    - **dict with keys:**
        * **set** - An ID of a dataset (or a `DataAnnotationSet` instance), only needed when specifying *key* as a string (see below)
        * **key** - A key, either an instance of *DataKey* or a string, in the latter case you need to specify *set* as well.

        * **value** or variants (see keyword arguments below)

    - ****kwargs** (*dict, optional*) –
        - **limit:** ***Optional[int] = None***
          The maximum number of results to return (default: unlimited)
        - **set:** ***Optional[Union[str,AnnotationDataSet]] = None***
          An ID of a dataset (or an `AnnotationDataSet` instance), only needed when specifying *key* as a string
        - **key:** ***Optional[Union[str,DataKey]] = None***
          An ID of a key (or a `DataKey` instance), make sure to specify *set* as well if you use a string value for this parameter.
        - **value:** ***Optional[Union[str,int,float,bool,List[Union[str,int,float,bool]]]]***
          Search for data matching a specific value. This holds exact value to search for. Further variants of this keyword are listed below:
        - **value_not:** ***Optional[Union[str,int,float,bool]]***
          Value must not match
        - **value_greater:** ***Optional[Union[int,float]]***
          Value must be greater than specified (int or float)
        - **value_less:** ***Optional[Union[int,float]]***
          Value must be less than specified (int or float)
        - **value_greatereq:** ***Optional[Union[int,float]]***
          Value must be greater than specified or equal (int or float)
        - **value_lesseq:** ***Optional[Union[int,float]]***
          Value must be less than specified or equal (int or float)
        - **value_in:** ***Optional[Tuple[Union[str,int,float,bool]]]***
          Value must match any in the tuple (this is a logical OR statement)
        - **value_not_in:** ***Optional[Tuple[Union[str,int,float,bool]]]***
          Value must not match any in the tuple
        - **value_in_range:** ***Optional[Tuple[Union[int,float]]]***
          Must be a numeric 2-tuple with min and max (inclusive) values

Return type
Data

dataset(*id: str*) → *AnnotationDataSet*

Basic retrieval method that returns an `AnnotationDataSet` by ID. Raises an exception if not found.

Parameters
id (*str*) –

Return type
AnnotationDataSet

datasets() → Iterator[*AnnotationDataSet*]

Returns an iterator over all annotation data sets (`AnnotationDataSet`) in this store

Return type
Iterator[*AnnotationDataSet*]

datasets_len() → int

Returns the number of annotation data sets in the store (not substracting deletions)

Return type
int

id() → str | None

Returns the public identifier (by value, aka a copy)

Return type
Optional[str]

key(*set_id: str*, *key_id: str*) → *DataKey*

Shortcut retrieval method that returns an *DataKey* by ID. Raises an exception if not found.

Parameters

- **set_id** (*str*) –
- **key_id** (*str*) –

Return type
DataKey

query(*query: str*, ***kwargs*) → list

Query the data using STAMQL.

Parameters

- **query** (*str*) – Query in STAMQL. Note that you *MUST* specify a variable to bind to in your *SELECT* statement (this is normally optional but is required for calling from Python).
- ****kwargs** (*tuple, optional*) – You can bind extra context variables using keyword arguments. The keys correspond to the variable names that these will be bound to and which you can subsequently use in the STAMQL query. These keys should not carry the '?' prefix you may be accustomed to in STAMQL. The value must be instances of STAM objects such as *Annotation*, *AnnotationData*, *DataKey*, :class`TextSelection` etc. These context variables are available to the query but not propagated to the output.

Return type
list

A query returns a list consisting of dictionaries, each corresponding one result row. The keys in the dictionaries match with the variable names in the STAMQL query, the values are result instances of whatever type the query returns, i.e. Annotation, AnnotationData, TextResource, TextSelection, AnnotationDataSet.

Examples

Query for annotations with certain kind of data:

```
for row in store.query('SELECT ANNOTATION ?a WHERE "some-set" "pos" = "noun";
↪'):
    for result in row:
        #just print out the text of the annotation
        print(str(result['a']))
```

resource(*id: str*) → *TextResource*

Basic retrieval method that returns a *TextResource* by ID. Raises an exception if not found.

Parameters
id (*str*) –

Return type
TextResource

resources() → Iterator[*TextResource*]

Returns an iterator over all text resources (*TextResource*) in this store

Return type
Iterator[*TextResource*]

resources_len() → int

Returns the number of text resources in the store (not substracting deletions)

Return type
int

save() → None

Saves the annotation store to the same file it was loaded from or last saved to.

Return type
None

set_filename(*filename: str*) → None

Set the filename for the annotationstore, the format is derived from the extension, can be *.json* or *csv*

Parameters
filename (*str*) –

Return type
None

shrink_to_fit()

Reallocates internal data structures to tight fits to conserve memory space (if necessary). You can use this after having added lots of annotations to possibly reduce the memory consumption.

to_file(*filename: str*) → None

Saves the annotation store to file. Use either .json or .csv as extension.

Parameters
filename (*str*) –

Return type
None

to_json_string() → str

Returns the annotation store as one big STAM JSON string

Return type
str

class stam.**Annotations**

An *Annotations* object holds an arbitrary collection of annotations. The annotations are references to items in an AnnotationStore, not copies. You can iterate over it to retrieve *Annotation* instances.

__getitem__(*int*) → *Annotation*

Returns an annotation in the collection by index

Return type
Annotation

__iter__() → Iterator[*Annotation*]

Iterator over all annotations in this collection

Return type
Iterator[*Annotation*]

__len__() → int

Returns the number of annotations in the collection

Return type
int

annotations(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation*) that reference annotations in the current collection (e.g. annotations that target of the current any annotations using an AnnotationSelector).

The annotations can be filtered using positional and/or keyword arguments; see *Annotation.annotations()*. If no filters are set (default), all annotations are returned (without duplicates) in chronological order.

Example

Say *annotation* represents a word, we can get all annotations that with key "part-of-speech", that point to this annotation:

```
key = store.dataset("linguistic-set").key("part-of-speech")
for pos_annotation in annotation.annotations(filter=key):
    data = annotation.data(filter=key,limit=1)[0]
    ...
```

Return type
Annotations

annotations_in_targets(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation*) that are being referenced by annotations in the current collection (e.g. annotations we target using an AnnotationSelector).

The annotations can be filtered using positional and/or keyword arguments; see *Annotation.annotations()*. One extra keyword argument is available and explained below. If no filters are set (default), all annotations are returned (without duplicates). Annotations are returned in chronological order.

Keyword Arguments
recursive (*bool*) – Follow AnnotationSelectors recursively (default False)

Return type
Annotations

data(**args*, ***kwargs*) → *Data*

Returns annotation data (*Data* containing *AnnotationData*) used by annotations in this collection.

The data can be filtered using positional and/or keyword arguments; see *Annotation.data()*. If no filters are set (default), all data from all annotations are returned (without duplicates).

Return type
Data

is_sorted() → bool

Returns a boolean indicating whether the annotations in this collection are sorted chronologically (earlier annotations before later once). Note that this is distinct from any textual ordering.

Return type
bool

related_text(*operator:* TextSelectionOperator, ***kwargs*) → *TextSelections*

Applies a *TextSelectionOperator* to find all other text selections who are in a specific relation with any from the current collection of annotations. Returns a collection of all matching *TextSelection* instances.

Text selections will be returned in textual order. They may be filtered via keyword arguments. See *Annotation.textselections()*.

See *Annotation.related_text()* for allowed paramters/keyword arguments and examples.

Parameters
operator (TextSelectionOperator) –

Return type
TextSelections

test_annotation(**args*, ***kwargs*) → bool

Tests whether certain annotations reference any annotation in this collection. The annotation can be filtered using positional and/or keyword arguments. See *annotations()*. Unlike *annotations()*, this method merely tests without returning the data, and as such is more performant.

Return type
bool

test_annotations_in_targets(**args*, ***kwargs*) → *Annotations*

Tests whether annotations in this collection targets the specified annotation. The annotation can be filtered using positional and/or keyword arguments. See *annotations()*. Unlike *annotations_in_targets()*, this method merely tests without returning the data, and as such is more performant.

Return type
Annotations

test_data(**args*, ***kwargs*) → bool

Tests whether certain annotation data is used by any annotation in this collection. The data can be filtered using keyword arguments. See *data()*. Unlike *data()*, this method merely tests without returning the data, and as such is more performant.

Return type
bool

textselections(*limit: int | None = None*) → *TextSelections*

Returns a collection of all textselections associated with the annotations in this collection.

Parameters
limit (*Optional[int]*) –

Return type
TextSelections

textual_order() → *Annotations*

Sorts the annotations in textual order (provided they refer to any text at all)

This has some performance cost, so prevent calling this method on methods like *Annotation.annotations_in_targets()* which already produce textual order (in most cases)

Return type
Annotations

class stam.**Cursor**(*index*, *endaligned: bool = False*)

A cursor points to a specific point in a text. It is used to select offsets. Units are unicode codepoints (not bytes!) and are 0-indexed.

The cursor can be either begin-aligned or end-aligned. Where BeginAlignedCursor(0) is the first unicode codepoint in a referenced text, and EndAlignedCursor(0) the last one.

Parameters

- **index** (*int*) – The value for the cursor.
- **endaligned** (*bool*) – Signals you want an end-aligned cursor, otherwise it is begin-aligned. If set this to True the index value should be 0 or negative, otherwise 0 or positive.

__str__() → str

Get a string representation of the cursor

Return type
str

is_beginaligned() → bool

Tests if this is a begin-aligned cursor

Return type
bool

is_endaligned() → bool

Tests if this is an end-aligned cursor

Return type
bool

value() → int

Get the actual cursor value

Return type
int

class stam.**Data**

A *Data* object holds an arbitrary collection of annotation data. The data are references to items in an AnnotationStore, not copies. You can iterate over it to retrieve *AnnotationData* instances.

__getitem__(*int*) → *AnnotationData*

Returns data in the collection by index

Return type
AnnotationData

__iter__() → Iterator[*AnnotationData*]

Iterator over all data in this collection

Return type
Iterator[*AnnotationData*]

__len__() → int

Returns the number of data items in the collection

Return type
int

annotations(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation*) that are make use of any of the data in this collection

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
Annotations

test_annotations(**args*, ***kwargs*) → bool

Tests whether there are any annotations that make use of any of the data in this collection This method is like *annotations()*, but does only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
bool

class stam.**DataKey**

The DataKey class defines a vocabulary field, it belongs to a certain *AnnotationDataSet*. A *AnnotationData* instance in turn makes reference to a DataKey and assigns it a value.

annotations(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation*) that make use of this key.

The annotations can be filtered on value using keyword arguments. See *Annotation.annotations()*, but note that not all keyword arguments apply in this context (set and key are predetermined already).

Example

Assume the key represents part-of-speech tags, get all annotations for value "noun":

```
for annotation in key.annotations(value="noun"):
    ...
```

Return type
Annotations

annotations_count(*limit: int | None = None*) → int

Returns the number of annotations (*Annotation*) that use this data. Note that this is much faster than doing *len(annotations())*! This method has suffix *_count* instead of *_len* because it is not O(1) but does actual counting (O(n) at worst).

Parameters
limit (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)

Return type
int

data(**args*, ***kwargs*) → *Data*

Returns annotation data (*Data* containing *AnnotationData*) used by this key.

The data can be filtered using positional and/or keyword arguments. See *Annotation.data()*. Note that only a subset makes sense in this context, set and key are already fixed.

Example

Assume the key represents part-of-speech tags, get all annotations for value "noun":

```
for data in key.data(value="noun"):
    # returns only one
```

Return type
Data

dataset() → *AnnotationDataSet*

Returns the *AnnotationDataSet* this key is part of

Return type
AnnotationDataSet

has_id(*id: str*) → bool

Tests the ID

Parameters
id (*str*) –

Return type
bool

id() → str | None

Returns the public ID (by value, aka a copy) Don't use this for extensive ID comparisons, use *has_id()* instead as it is more performant (no copy).

Return type
Optional[str]

select() → *Selector*

Returns a selector pointing to this key (DataKeySelector)

Return type
Selector

test_annotations(**args*, ***kwargs*) → bool

Tests whether there are any annotations that make use of this key. This method is like *annotations()*, but only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using keyword arguments. See *Annotation.annotations()*.

Example

Assume the key represents part-of-speech tags, test if there are annotations for data value "noun":

if key.test_annotations(value="noun"):
    ...

Return type
bool

test_data(**args*, ***kwargs*) → bool

Tests whether certain annotation data exists for this key The data can be filtered using keyword arguments. See *Annotation.data()*. Note that only a subset makes sense in this context, set and key are already fixed.

This method is like *data()*, but merely tests without returning the data, and as such is more performant.

Example

Assume the key represents part-of-speech tags, get all annotations for value "noun":

```
if key.test_data(value="noun"):
    #value exists
    ...
```

Return type
bool

class stam.**DataValue**(*value: str | bool | int | float | List*)

Encapsulates a value and its type. Held by *AnnotationData*. This type is not a reference but holds the actual value.

You can instantiate a new DataValue from a supported Python type, but you usually don't need to do this explicitly.

Parameters
value (*Union[str, bool, int, float, List]*) –

__str__() → str

Get the actual value as as string

Return type
str

get() → str | bool | int | float | List

Get the actual value

Return type
Union[str, bool, int, float, List]

class stam.**Offset**(*begin:* Cursor, *end:* Cursor)

Text selection offset. Specifies begin and end offsets to select a range of a text, via two *Cursor* instances. The end-point is non-inclusive.

You can instantiate a new offset on the basis of two *Cursor* instances

Parameters

- **begin** (Cursor) –
- **end** (Cursor) –

__str__() → str

Get a string representation of the offset

Return type
str

begin() → *Cursor*

Returns the begin cursor

Return type
Cursor

end() → *Cursor*

Returns the end cursor

Return type
Cursor

static simple(*begin: int*, *end: int*) → *Offset*

Instantiate a new offset on the basis of two begin aligned cursors

Parameters

- **begin** (*int*) –
- **end** (*int*) –

Return type
Offset

static whole() → *Offset*

Instantiate a new offset that targets an entire text from begin to end.

Return type
Offset

class stam.**Selector**

A *Selector* identifies the target of an annotation and the part of the target that the annotation applies to. Selectors can be considered the labelled edges of the graph model, tying all nodes together. There are multiple types of selectors, all captured in this class. There are several static methods available to instantiate a specific type of selector.

annotation(*store:* AnnotationStore) → *Annotation* | None

Returns the annotation this selector points at, if any. Works only for AnnotationSelector, returns None otherwise. Requires to explicitly pass the store so the resource can be found.

Parameters
store (AnnotationStore) –

Return type
Optional[*Annotation*]

static annotationselector(*annotation:* Annotation, *offset:* Offset | *None* = *None*) → *Selector*

Creates an *AnnotationSelector* - A selector pointing to another annotation. This we call higher-order annotation and is very common in STAM models. If the annotation that is being targeted eventually refers to a text (*TextSelector*), then offsets **MAY** be specified that select a subpart of this text. These offsets are now *relative* to the annotation.

Parameters

- **annotation** (Annotation) – The target annotation

- **offset** (*Optional[Offset]*) – If sets, references a subpart of the annotation's text. If set to *None*, it applies to the annotation as such.

Return type
Selector

Example

Instantiation:

```
Selector.textselector(store.annotation("A1"), Offset.whole())
```

static compositeselector(**subselectors:* Selector) → *Selector*

Creates a *CompositeSelector* - A selector that consists of multiple other selectors (subselectors), these are used to select more complex targets that transcend the idea of a single simple selection. This *MUST* be interpreted as the annotation applying equally to the conjunction as a whole, its parts being inter-dependent and for any of them it goes that they *MUST NOT* be omitted for the annotation to make sense.

Parameters
***subselectors** (Selector) – The underlying selectors.

Return type
Selector

Example

Instantiation of a composite selector over two annotation selectors:

```
Selector.compositeselector(
    Selector.annotationselector(self.store.annotation("A1"), Offset.whole()),
    Selector.annotationselector(self.store.annotation("A2"), Offset.whole()),
)
```

dataset(*store:* AnnotationStore) → *AnnotationDataSet* | None

Returns the annotation dataset this selector points at, ff any. Works only for DataSetSelector, returns None otherwise. Requires to explicitly pass the store so the dataset can be found.

Parameters
store (AnnotationStore) –

Return type
Optional[*AnnotationDataSet*]

static datasetselector(*dataset:* AnnotationDataSet) → *Selector*

Creates a *DataSetSelector* - A selector pointing to an annotation dataset as whole. These type of annotation can be interpreted as *metadata*.

Parameters
dataset (AnnotationDataSet) – The annotation data set.

Return type
Selector

Example

Instantiation:

```
Selector.datasetselector(store.dataset("my-dataset"))
```

static directionalselector(**subselectors:* Selector) → *Selector*

Creates a *DirectionalSelector* - Another selector that consists of multiple other selectors, but with an explicit direction (from -> to), used to select more complex targets that transcend the idea of a single simple selection.

Parameters
***subselectors** (Selector) – The underlying selectors.

Return type
Selector

is_kind(*kind:* SelectorKind) → bool

Tests whether a selector is of a particular type

Parameters
kind (SelectorKind) –

Return type
bool

kind() → *SelectorKind*

Returns the type of selector

Return type
SelectorKind

static multiselector(**subselectors:* Selector) → *Selector*

Creates a *MultiSelector* - A selector that consists of multiple other selectors (subselectors) to select multiple targets. This *MUST* be interpreted as the annotation applying to each target *individually*, without any relation between the different targets.

Parameters
***subselectors** (Selector) – The underlying selectors.

Return type
Selector

offset() → *Offset* | None

Return offset information in the selector. Works for TextSelector and AnnotationSelector, returns None for others.

Return type
Optional[*Offset*]

resource(*store:* AnnotationStore) → *TextResource* | None

Returns the resource this selector points at, if any. Works only for TextSelector and ResourceSelector, returns None otherwise. Requires to explicitly pass the store so the resource can be found.

Parameters
store (AnnotationStore) –

Return type
Optional[*TextResource*]

static resourceselector(*resource:* TextResource) → *Selector*

Creates a *ResourceSelector* - A selector pointing to a resource as whole. These type of annotation can be interpreted as *metadata*.

Parameters
resource (TextResource) – The resource

Return type
Selector

Example

Instantiation:

```
Selector.resourceselector(store.resource("my-resource"))
```

static textselector(*resource:* TextResource, *offset:* Offset) → *Selector*

Creates a *TextSelector*. Selects a target resource and a text span within it.

Parameters

- **resource** (TextResource) – The text resource
- **offset** (Offset) – An offset pointing to the slice of the text in the resource

Return type
Selector

Example

Instantiation:

```
Selector.textselector(store.resource("testres"), Offset.simple(6,11))
```

class stam.**SelectorKind**

An enumeration of possible selector types

ANNOTATIONDATASELECTOR: *SelectorKind*

ANNOTATIONSELECTOR: *SelectorKind*

COMPOSITESELECTOR: *SelectorKind*

DATAKEYSELECTOR: *SelectorKind*

DATASETSELECTOR: *SelectorKind*

DIRECTIONALSELECTOR: *SelectorKind*

MULTISELECTOR: *SelectorKind*

RESOURCESELECTOR: *SelectorKind*

TEXTSELECTOR: *SelectorKind*

exception stam.**StamError**

Bases: Exception

STAM Error

Initialize self. See help(type(self)) for accurate signature.

class stam.**TextResource**

This holds the textual resource to be annotated. It holds the full text in memory.

The text *SHOULD* be in [Unicode Normalization Form C (NFC) (https://www.unicode.org/reports/tr15/) but *MAY* be in another unicode normalization forms.

__getitem__(*slice: TextResource.__getitem__.slice*) → str

Returns a text slice

Parameters
slice (*TextResource.__getitem__.slice*) –

Return type
str

__iter__() → Iterator[*TextSelection*]

Iterates over all known textselections in this resource, in sorted order. This is a low-level iterator, *textselections()* provides a higher-level interface.

Return type
Iterator[*TextSelection*]

__str__() → str

Returns the text of the resource (by value, aka a copy), same as *text()*

Return type
str

annotations(**args*, ***kwargs*) → *Annotations*

Returns a collection of annotations (*Annotation*) that reference this resource via a *TextSelector* (if any). Does *NOT* include those that use a ResourceSelector, use annotations_metadata() instead for those instead.

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
Annotations

annotations_as_metadata(**args*, ***kwargs*) → *Annotations*

Returns a collection of annotations (*Annotation*) that reference this resource via a *ResourceSelector* (if any). Does *NOT* include those that use a TextSelector, use *annotations()* instead for those instead.

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
Annotations

beginaligned_cursor(*endalignedcursor: int*) → int

Converts an end-aligned cursor to a begin-aligned cursor, resolving all relative end-aligned positions The parameter value must be 0 or negative.

Parameters
endalignedcursor (*int*) –

Return type
int

find_text(*fragment: str*, *limit: int | None = None*, *case_sensitive: bool | None = None*) → List[*TextSelection*]

Searches for the text fragment and returns a list of `TextSelection` instances with all matches (or up to the specified limit)

Parameters

- **fragment** (*str*) – The exact fragment to search for (case-sensitive)
- **limit** (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)
- **case_sensitive** (*Optional[bool] = None*) – Match case sensitive or not (default: True)

Return type
List[*TextSelection*]

find_text_regex(*expressions: List[str]*, *allow_overlap: bool | None = False*, *limit: int | None = None*) → List[dict]

Searches the text using one or more regular expressions, returns a list of dictionaries like:

code:

```
{ "textselections": [TextSelection], "expression_index": int, "capturegroups":␣
↪[int] }
```

Passing multiple regular expressions at once is more efficient than calling this function anew for each one. If capture groups are used in the regular expression, only those parts will be returned (the rest is context). If none are used, the entire expression is returned. The regular expressions are passed as strings and must follow this syntax: https://docs.rs/regex/latest/regex/#syntax , which may differ slightly from Python's regular expressions!

The *allow_overlap* parameter determines if the matching expressions are allowed to overlap. It you are doing some form of tokenisation, you also likely want this set to false. All of this only matters if you supply multiple regular expressions.

Results are returned in the exact order they are found in the text

Parameters

- **expressions** (*List[str]*) –
- **allow_overlap** (*Optional[bool]*) –
- **limit** (*Optional[int]*) –

Return type
List[dict]

has_id(*id: str*) → bool

Tests the ID

Parameters
id (*str*) –

Return type
bool

id() → str | None

Returns the public ID (by value, aka a copy) Don't use this for extensive ID comparisons, use *has_id()* instead as it is more performant (no copy).

Return type
Optional[str]

range(*begin*, *end*) → Iterator[*TextSelection*]

Iterates over all known textselections that start in the specified range, in sorted order.

Return type
Iterator[*TextSelection*]

select() → *Selector*

Returns a selector pointing to this resource

Return type
Selector

split_text(*delimiter: str*, *limit: int | None = None*) → List[*TextSelection*]

Returns a list of *TextSelection* instances that split the text according to the specified delimiter.

Parameters

- **delimiter** (*str*) – The delimiter to split on
- **limit** (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)

Return type
List[*TextSelection*]

strip_text(*chars: str*) → *TextSelection*

Trims all occurrences of any character in *chars* from both the beginning and end of the text, returning a *TextSelection*. No text is modified.

Parameters
chars (*str*) –

Return type
TextSelection

test_annotations(**args*, ***kwargs*) → bool

Tests whether there are any annotations that reference the text of this resource (via a TextSelector).

This method is like *annotations()*, but only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
bool

test_annotations_as_metadata(**args*, ***kwargs*) → bool

Tests whether there are any annotations that reference this resource as metadata (via a ResourceSelector).

This method is like *annotations_as_metadata()*, but only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
bool

text() → str

Returns the text of the resource (by value, aka a copy)

Return type
str

textlen() → int

Returns the length of the resources's text in unicode points (same as *len(self.text())* but more performant)

Return type
int

textselection(*offset:* Offset) → *TextSelection*

Returns a *TextSelection* instance covering the specified offset.

Parameters
offset (*Offset*) –

Return type
TextSelection

textselections() → *TextSelections*

Iterates over all known textselections in this resource, in sorted order.

Return type
TextSelections

utf8byte(*abscursor: int*) → int

Converts a unicode character position to a UTF-8 byte position

Parameters
abscursor (*int*) –

Return type
int

utf8byte_to_charpos(*bytecursor: int*) → int

Converts a UTF-8 byte position into a unicode position

Parameters
bytecursor (*int*) –

Return type
int

class stam.**TextSelection**

This holds a slice of a text.

__getitem__(*slice: TextSelection.__getitem__.slice*) → str

Returns a text slice

Parameters
slice (*TextSelection.__getitem__.slice*) –

Return type
str

__str__() → str

Returns the text of the resource (by value, aka a copy), same as *text()*

Return type
str

annotations(**kwargs) → *Annotations*

Returns annotations (*Annotations* containing *Annotation*) that reference this text selection via a *TextSelector* (if any).

The annotations can be filtered using keyword arguments. See *Annotation.annotations()*

Return type
Annotations

annotations_len() → int

Returns the number of annotations this text selection references

Return type
int

begin() → int

Return the absolute begin position in unicode points

Return type
int

beginaligned_cursor(*endalignedcursor: int*) → int

Converts an end-aligned cursor to a begin-aligned cursor, resolving all relative end-aligned positions The parameter value must be 0 or negative.

Parameters
endalignedcursor (*int*) –

Return type
int

end() → int

Return the absolute end position in unicode points (non-inclusive)

Return type
int

find_text(*fragment: str*, *limit: int | None = None*, *case_sensitive: bool | None = None*) → List[*TextSelection*]

Searches for the text fragment and returns a list of *TextSelection* instances with all matches (or up to the specified limit)

Parameters

- **fragment** (*str*) – The exact fragment to search for
- **limit** (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)
- **case_sensitive** (*Optional[bool] = None*) – Match case sensitive or not (default: True)

Return type
List[*TextSelection*]

find_text_regex(*expressions: List[str]*, *allow_overlap: bool | None = False*, *limit: int | None = None*) → List[dict]

Searches the text using one or more regular expressions, returns a list of dictionaries like:

code:

```
{ "textselections": [TextSelection], "expression_index": int, "capturegroups":␣
↪[int] }
```

Passing multiple regular expressions at once is more efficient than calling this function anew for each one. If capture groups are used in the regular expression, only those parts will be returned (the rest is context). If none are used, the entire expression is returned. The regular expressions are passed as strings and must follow this syntax: https://docs.rs/regex/latest/regex/#syntax , which may differ slightly from Python's regular expressions!

The *allow_overlap* parameter determines if the matching expressions are allowed to overlap. It you are doing some form of tokenisation, you also likely want this set to false. All of this only matters if you supply multiple regular expressions.

Results are returned in the exact order they are found in the text

Parameters

- **expressions** (*List[str]*) –
- **allow_overlap** (*Optional[bool]*) –
- **limit** (*Optional[int]*) –

Return type
List[dict]

find_text_sequence(*fragments: List[str]*, *case_sensitive: bool | None = None*, *allow_skip_whitespace: bool | None = True*, *allow_skip_punctuation: bool | None = True*, *allow_skip_numeric: bool | None = True*, *allow_skip_alphabetic: bool | None = False*) → List[*TextSelection*]

Searches for the multiple text fragment in sequence. Returns a list of *TextSelection* instances.

Matches must appear in the exact order specified, but *may* have other intermittent text, determined by the *allow_skip_** parameters.

Returns an empty list if the sequence does not match.

Parameters

- **fragments** (*List[str]*) – The fragments to search for, in sequence
- **case_sensitive** (*Optional[bool] = None*) – Match case sensitive or not (default: True)
- **allow_skip_whitespace** (*Optional[bool] = True*) – Allow gaps consisting of whitespace (space, tabs, newline, etc) (default: True)
- **allow_skip_punctuation** (*Optional[bool] = True*) – Allow gaps consisting of punctuation (default: True)
- **allow_skip_numeric** (*Optional[bool] = True*) – Allow gaps consisting of numbers (default: True)
- **allow_skip_alphabetic** (*Optional[bool] = True*) – Allow gaps consisting of alphabetic/ideographic characters (default: False)

Return type
List[*TextSelection*]

related_text(*operator:* TextSelectionOperator, ***kwargs*) → *TextSelections*

Applies a *TextSelectionOperator* to find all other text selections who are in a specific relation with this one. Returns all matching *TextSelection* instances in a collection *TextSelections*.

Text selections will be returned in textual order. They may be filtered via keyword arguments. See *Annotation.textselections()*.

Parameters
operator (TextSelectionOperator) – The operator to apply when comparing text selections

Return type
TextSelections

See *Annotation.related_text()* for allowed keyword arguments and examples.

relative_offset(*container:* TextSelection) → *Offset*

Returns the offset of this text selection relative to another in which it is *embedded*. Raises a *StamError* exception if they are not embedded, or not belonging to the same resource.

Parameters
container (TextSelection) –

Return type
Offset

resource() → *TextResource*

Returns the *TextResource* this textselection is from.

Return type
TextResource

select() → *Selector*

Returns a selector pointing to this resource

Return type
Selector

split_text(*delimiter: str*, *limit: int | None = None*) → List[*TextSelection*]

Returns a list of *TextSelection* instances that split the text according to the specified delimiter.

Parameters

- **delimiter** (*str*) – The delimiter to split on
- **limit** (*Optional[int] = None*) – The maximum number of results to return (default: unlimited)

Return type
List[*TextSelection*]

strip_text(*chars: str*) → *TextSelection*

Trims all occurrences of any character in *chars* from both the beginning and end of the text, returning a *TextSelection*. No text is modified.

Parameters
chars (*str*) –

Return type
TextSelection

test(*operator:* TextSelectionOperator, *other:* TextSelection) → bool

This method is called to test whether a specific spatial relation (as expressed by the passed operator) holds between a [*TextSelection*] and another. A boolean is returned with the test result.

Parameters

- **operator** (TextSelectionOperator) –
- **other** (TextSelection) –

Return type
bool

test_annotations(**kwargs*) → bool

Tests whether there are any annotations that reference this text selection via a *TextSelector* (if any).

This method is like *annotations()*, but only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using keyword arguments. See *Annotation.annotations()*.

Return type
bool

test_data(**kwargs*) → bool

Tests whether there are any annotations that reference this text selection with data that passes the provided filters. The result is functionally equivalent to doing *.annotations().test_data()*, but this shortcut method is implemented much more efficiently and therefore recommended.

The data can be filtered using keyword arguments. See *Annotations.data()*.

Return type
bool

text() → str

Returns the text of the resource (by value, aka a copy)

Return type
str

textlen() → int

Returns the length of the resources's text in unicode points (same as *len(self.text())* but more performant)

Return type
int

textselection(*offset:* Offset) → *TextSelection*

Returns a *TextSelection* that corresponds to the offset **WITHIN** the current textselection. This returns a *TextSelection* with absolute coordinates in the resource.

Parameters
offset (Offset) –

Return type
TextSelection

utf8byte(*abscursor: int*) → int

Converts a unicode character position to a UTF-8 byte position

Parameters
abscursor (*int*) –

Return type
int

utf8byte_to_charpos(*bytecursor: int*) → int

Converts a UTF-8 byte position into a unicode position

Parameters
bytecursor (*int*) –

Return type
int

class stam.**TextSelectionOperator**

The TextSelectionOperator, simply put, allows comparison of two *TextSelection* instances. It allows testing for all kinds of spatial relations (as embodied by this class) in which two *TextSelection* instances can be.

Rather than operate on single *TextSelection* instances, the implementation goes a bit further and can act also on the basis of multiple *TextSelection* instances as a set; allowing you to compare two sets, each containing possibly multiple TextSelections, at once.

The operator is instantiated via one of its static methods.

static after(*all: bool | None = False*, *negate: bool | None = False*, *limit: int | None = None*) → *TextSelectionOperator*

Create an operator to test if one textselection(sets) comes after another Each TextSeleciton In A comes after a textselection in B If modifier *all* is set: All TextSelections in A come after all textselections in B. There is no overlap (cf. textfabric's >>)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).
- **limit** (*Optional[usize]*) – Constrain the lookup to at most this many unicode points (increases performance)

Return type
TextSelectionOperator

static before(*all: bool | None = False*, *negate: bool | None = False*, *limit: int | None = None*) → *TextSelectionOperator*

Create an operator to test if one textselection(sets) comes before another Each TextSelections in A comes before a textselection in B If modifier *all* is set: All TextSelections in A come before all textselections in B. There is no overlap (cf. textfabric's <<)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).
- **limit** (*Optional[usize]*) – Constrain the lookup to at most this many unicode points (increases performance)

Return type
TextSelectionOperator

static **embedded**(*all: bool | None = False*, *negate: bool | None = False*, *limit: int | None = None*) → *TextSelectionOperator*

Create an operator to test if two textselection(sets) are embedded. All TextSelections in B are embedded by a TextSelection in A (cf. textfabric's *[]*) If modifier *all* is set: All TextSelections in B are embedded by all TextSelection in A (cf. textfabric's *[]*)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).
- **limit** (*Optional[usize]*) – Constrain the lookup to at most this many unicode points (increases performance)

Return type
TextSelectionOperator

static **embeds**(*all: bool | None = False*, *negate: bool | None = False*) → *TextSelectionOperator*

Create an operator to test if two textselection(sets) are embedded. All TextSelections in B are embedded by a TextSelection in A (cf. textfabric's *[]*) If modifier *all* is set: All TextSelections in B are embedded by all TextSelection in A (cf. textfabric's *[]*)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).

Return type
TextSelectionOperator

static **equals**(*all: bool | None = False*, *negate: bool | None = False*) → *TextSelectionOperator*

Create an operator to test if two textselection(sets) occupy cover the exact same TextSelections, and all are covered (cf. textfabric's ==), commutative, transitive

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).

Return type
TextSelectionOperator

static **overlaps**(*all: bool | None = False*, *negate: bool | None = False*) → *TextSelectionOperator*

Create an operator to test if two textselection(sets) overlap. Each TextSelection in A overlaps with a TextSelection in B (cf. textfabric's *&&*), commutative If modifier *all* is set: Each TextSelection in A overlaps with all TextSelection in B (cf. textfabric's *&&*), commutative

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).

- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).

Return type
TextSelectionOperator

static precedes(*all: bool | None = False*, *negate: bool | None = False*) → *TextSelectionOperator*

Create an operator to test if one textselection(sets) is to the immediate left (precedes) of another Each TextSelection in A is ends where at least one TextSelection in B begins. If modifier *all* is set: The rightmost TextSelections in A end where the leftmost TextSelection in B begins (cf. textfabric's *<:*)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).

Return type
TextSelectionOperator

static samebegin(*all: bool | None = False*, *negate: bool | None = False*) → *TextSelectionOperator*

Create an operator to test if two textselection(sets) have the same begin position Each TextSelection in A starts where a TextSelection in B starts If modifier *all* is set: The leftmost TextSelection in A starts where the leftmost TextSelection in B start (cf. textfabric's *=:*)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).

Return type
TextSelectionOperator

static sameend(*all: bool | None = False*, *negate: bool | None = False*) → *TextSelectionOperator*

Create an operator to test if two textselection(sets) have the same end position Each TextSelection in A ends where a TextSelection in B ends If modifier *all* is set: The rightmost TextSelection in A ends where the rights TextSelection in B ends (cf. textfabric's *:=*)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).
- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).

Return type
TextSelectionOperator

static succeeds(*all: bool | None = False*, *negate: bool | None = False*) → *TextSelectionOperator*

Create an operator to test if one textselection(sets) is to the immediate right (succeeds) of another Each TextSelection in A is begis where at least one TextSelection in A ends. If modifier *all* is set: The leftmost TextSelection in A starts where the rightmost TextSelection in B ends (cf. textfabric's *:>*)

Parameters

- **all** (*Optional[bool]*) – If this is set, then for each *TextSelection* in A, the relationship must hold with **ALL** of the text selections in B. The normal behaviour, when this is set to false, is a match with any item suffices (and may be returned).

- **negate** (*Optional[bool]*) – Inverses the operator (turns it into a negation).

Return type
TextSelectionOperator

class stam.**TextSelections**

A *TextSelections* object holds an arbitrary collection of text selections. You can iterate over it to retrieve *TextSelection* instances.

__getitem__(*int*) → *TextSelection*

Returns a textselection in the collection by index

Return type
TextSelection

__iter__() → Iterator[*TextSelection*]

Iterator over all text selections in this collection

Return type
Iterator[*TextSelection*]

__len__() → int

Returns the number of data items in the collection

Return type
int

__str__() → str

Returns the text of all textselections.

The results are space-delimited, use *text_join()* instead if you want another delimiter.

Return type
str

annotations(**args*, ***kwargs*) → *Annotations*

Returns annotations (*Annotations* containing *Annotation*) that refer to any of the text selections in this collection

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
Annotations

data(**args*, ***kwargs*) → *Data*

Returns annotation data (*Data* containing *AnnotationData*) used by annotations referring to the text selections in this collection.

The data can be filtered using positional and/or keyword arguments; see *Annotation.data()*. If no filters are set (default), all data from all annotations on all text selections are returned (without duplicates).

Return type
Data

related_text(*operator:* TextSelectionOperator, **args*, ***kwargs*) → *TextSelections*

Applies a *TextSelectionOperator* to find all other text selections who are in a specific relation with the ones from the current collections. Returns a collection of all matching *TextSelection* instances.

Text selections will be returned in textual order. They may be filtered via positional and/or keyword arguments. See *Annotation.textselections()*.

If you are interested in the annotations associated with the found text selections, then add *.annotations()* to the result.

See *Annotation.related_text()* for allowed keyword arguments and examples.

Parameters
operator (TextSelectionOperator) –

Return type
TextSelections

test_annotations(**kwargs) → bool

Tests whether there are any annotations that refer to any of the text selections in this collection

This method is like *annotations()*, but only tests and does not return the annotations, as such it is more performant.

The annotations can be filtered using positional and/or keyword arguments. See *Annotation.annotations()*.

Return type
bool

test_data(*args, **kwargs) → bool

Tests whether there are any annotations that reference any of the text selections in the iterator, with data that passes the provided filters. The result is functionally equivalent to doing *.annotations().test_data()*, but this shortcut method is implemented much more efficiently and therefore recommended.

The data can be filtered using positional and/or keyword arguments. See *Annotations.data()*.

Return type
bool

text(*delimiter: str*) → List[str]

Returns the text of all textselections in a list

Parameters
delimiter (*str*) –

Return type
List[str]

text_join(*delimiter: str*) → str

Returns the text of all textselections, separated by the provider delimiter. This is more efficient than calling *.text().join()* yourself.

Parameters
delimiter (*str*) –

Return type
str

textual_order() → *TextSelections*

Sorts the annotations in textual order.

This has some performance cost, so prevent calling this method on methods that already promise to return textual order (which most textselection methods do!)

Return type
TextSelections

CHAPTER

THREE

INDEX

- genindex

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

M

O

P

Q

R

S

T

U

V

W