Ypy

Release 0.5.5

Kevin Jahns, Bartosz Sypytkowski, John Waidhofer

Sep 13, 2023

CONTENTS:

Ypy is a high-performance CRDT that allows Python developers to easily synchronize state between processes. It is built on top of Y-CRDT: a powerful distributed data type library written in Rust. With Ypy, developers can make robust, eventually consistent applications that share state between users. All changes are automatically resolved across application instances, so your code can focus on representing state instead of synchronizing it. This shared state can go beyond Python programs, interfacing to web applications backed by Y-Wasm. This allows for seamless communication between frontend user interfaces and Python application logic.

CHAPTER

ONE

INSTALLATION

You can install Ypy from PyPI:

```
pip install y-py
```

Or from conda:

```
conda install -c conda-forge y-py
```

CHAPTER

TWO

TUTORIAL

Each user working with Ypy data can read and update information through a shared document instance. Anything added to the document will be tracked and synchronized across all document instances. These documents can hold common data types, including numbers, booleans, strings, lists, dictionaries, and XML trees. Modifying the document state is done inside a transaction for robustness and thread safety. With these building blocks, you can safely share data between users. Here is a basic hello world example:

```
import y_py as Y

d1 = Y.YDoc()
# Create a new YText object in the YDoc
text = d1.get_text('test')
# Start a transaction in order to update the text
with d1.begin_transaction() as txn:
    # Add text contents
    text.extend(txn, "hello world!")

# Create another document
d2 = Y.YDoc()
# Share state with the original document
state_vector = Y.encode_state_vector(d2)
diff = Y.encode_state_as_update(d1, state_vector)
Y.apply_update(d2, diff)

value = str(d2.get_text('test'))

assert value == "hello world!"
```

CHAPTER

THREE

API REFERENCE

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

3.1 y_py

3.1.1 Module Contents

[1] Created with sphinx-autoapi

Classes

SubscriptionId	Tracks an observer callback. Pass this to the *unobserve* method to cancel
YDoc	A Ypy document type. Documents are most important units of collaborative resources management.
AfterTransactionEvent	Holds transaction update information from a commit after state vectors have been compressed.
YTransaction	A transaction that serves as a proxy to document block store. Ypy shared data types execute
YText	A shared data type used for collaborative text editing. It enables multiple users to add and
YTextEvent	Communicates updates that occurred during a transaction for an instance of *YText*.
YTextChangeInsert	
YTextChangeDelete	
YTextChangeRetain	
YArray	
YArrayEvent	Communicates updates that occurred during a transaction for an instance of *YArray*.
ArrayChangeInsert	Update message that elements were inserted in a YArray.
ArrayChangeDelete	Update message that elements were deleted in a YArray.
ArrayChangeRetain	Update message that elements were left unmodified in a YArray.
YMap	
YMapItemsView	Tracks key/values inside a YMap. Similar functionality to dict_items for a Python dict
YMapKeysView	Tracks key identifiers inside of a YMap
YMapValuesView	Tracks values inside of a YMap
YMapEvent	Communicates updates that occurred during a transaction for an instance of *YMap*.
YMapEventKeyChange	
YXmlElementEvent	
YXmlElement	XML element data type. It represents an XML node, which can contain key-value attributes
YXmlText	
YXmlTextEvent	

Functions

encode_state_vector(→ EncodedStateVector)	Encodes a state vector of a given Ypy document into its binary representation using lib0 v1
encode_state_as_update(→ YDocUpdate)	Encodes all updates that have happened since a given version *vector* into a compact delta
apply_update(doc, diff)	Applies delta update generated by the remote document replica to a current document. This

Attributes

Event	
EncodedStateVector	
EncodedDeleteSet	
YDocUpdate	
YTextDelta	
YArrayObserver	
ArrayDelta	A modification to a YArray during a transaction.
YXmlAttributes	Generates a sequence of key/value properties for an XML Element
Xml	
YXmlTreeWalker	Visits elements in an Xml tree
EntryChange	

class y_py.**SubscriptionId**

Tracks an observer callback. Pass this to the *unobserve* method to cancel its associated callback.

y_py.**Event**

class y_py.**YDoc**(*client_id: Optional[int] = None*, *offset_kind: str = 'utf8'*, *skip_gc: bool = False*)

A Ypy document type. Documents are most important units of collaborative resources management. All shared collections live within a scope of their corresponding documents. All updates are generated on per document basis (rather than individual shared type). All operations on shared collections happen via YTransaction, which lifetime is also bound to a document.

Document manages so called root types, which are top-level shared types definitions (as opposed to recursively nested types).

Example:

```
from y_py import YDoc

doc = YDoc()
```

(continues on next page)

(continued from previous page)

```
with doc.begin_transaction() as txn:
    text = txn.get_text('name')
    text.extend(txn, 'hello world')

print(str(text))
```

client_id: int

begin_transaction() → *YTransaction*

Returns
A new transaction for this document. Ypy shared data types execute their operations in a context of a given transaction. Each document can have only one active transaction at the time - subsequent attempts will cause exception to be thrown.

Transactions started with *doc.begin_transaction* can be released by deleting the transaction object method.

Example:

```
from y_py import YDoc
doc = YDoc()
text = doc.get_text('name')
with doc.begin_transaction() as txn:
    text.insert(txn, 0, 'hello world')
```

transact(*callback: Callable[[*YTransaction*]]*)

get_map(*name: str*) → *YMap*

Returns
A *YMap* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned.

If there was an instance with this name, but it was of different type, it will be projected onto *YMap* instance.

get_xml_element(*name: str*) → *YXmlElement*

Returns
A *YXmlElement* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned.

If there was an instance with this name, but it was of different type, it will be projected onto *YXmlElement* instance.

get_xml_text(*name: str*) → *YXmlText*

Returns
A *YXmlText* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned.

If there was an instance with this name, but it was of different type, it will be projected onto *YXmlText* instance.

get_array(*name: str*) → *YArray*

Returns
A *YArray* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned.

If there was an instance with this name, but it was of different type, it will be projected onto *YArray* instance.

get_text(*name: str*) → *YText*

Parameters
name – The identifier for retreiving the text

Returns
A *YText* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned. If there was an instance with this name, but it was of different type, it will be projected onto *YText* instance.

observe_after_transaction(*callback: Callable[[*AfterTransactionEvent*]]*) → *SubscriptionId*

Subscribe callback function to updates on the YDoc. The callback will receive encoded state updates and deletions when a document transaction is committed.

Parameters
callback – A function that receives YDoc state information affected by the transaction.

Returns
A subscription identifier that can be used to cancel the callback.

y_py.**EncodedStateVector**

y_py.**EncodedDeleteSet**

y_py.**YDocUpdate**

class y_py.**AfterTransactionEvent**

Holds transaction update information from a commit after state vectors have been compressed.

before_state: EncodedStateVector

Encoded state of YDoc before the transaction.

after_state: EncodedStateVector

Encoded state of the YDoc after the transaction.

delete_set: EncodedDeleteSet

Elements deleted by the associated transaction.

get_update() → YDocUpdate

Returns
Encoded payload of all updates produced by the transaction.

y_py.**encode_state_vector**(*doc:* YDoc) → EncodedStateVector

Encodes a state vector of a given Ypy document into its binary representation using lib0 v1 encoding. State vector is a compact representation of updates performed on a given document and can be used by *encode_state_as_update* on remote peer to generate a delta update payload to synchronize changes between peers.

Example:

```
from y_py import YDoc, encode_state_vector, encode_state_as_update, apply_update␣
↪from y_py

# document on machine A
local_doc = YDoc()
```

(continues on next page)

(continued from previous page)

```
local_sv = encode_state_vector(local_doc)

# document on machine B
remote_doc = YDoc()
remote_delta = encode_state_as_update(remote_doc, local_sv)

apply_update(local_doc, remote_delta)
```

y_py.**encode_state_as_update**(*doc: YDoc, vector: Optional[Union[EncodedStateVector, List[int]]] = None*) → YDocUpdate

Encodes all updates that have happened since a given version *vector* into a compact delta representation using lib0 v1 encoding. If *vector* parameter has not been provided, generated delta payload will contain all changes of a current Ypy document, working effectively as its state snapshot.

Example:

```
from y_py import YDoc, encode_state_vector, encode_state_as_update, apply_update

# document on machine A
local_doc = YDoc()
local_sv = encode_state_vector(local_doc)

# document on machine B
remote_doc = YDoc()
remote_delta = encode_state_as_update(remote_doc, local_sv)

apply_update(local_doc, remote_delta)
```

y_py.**apply_update**(*doc: YDoc, diff: Union[YDocUpdate, List[int]]*)

Applies delta update generated by the remote document replica to a current document. This method assumes that a payload maintains lib0 v1 encoding format.

Example:

```
from y_py import YDoc, encode_state_vector, encode_state_as_update, apply_update

# document on machine A
local_doc = YDoc()
local_sv = encode_state_vector(local_doc)

# document on machine B
remote_doc = YDoc()
remote_delta = encode_state_as_update(remote_doc, local_sv)

apply_update(local_doc, remote_delta)
```

class y_py.**YTransaction**

A transaction that serves as a proxy to document block store. Ypy shared data types execute their operations in a context of a given transaction. Each document can have only one active transaction at the time - subsequent attempts will cause exception to be thrown.

Transactions started with *doc.begin_transaction* can be released by deleting the transaction object method.

Example:

```
from y_py import YDoc
doc = YDoc()
text = doc.get_text('name')
with doc.begin_transaction() as txn:
    text.insert(txn, 0, 'hello world')
```

before_state: Dict[int, int]

get_text(*name: str*) → *YText*

Returns
A *YText* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned.

If there was an instance with this name, but it was of different type, it will be projected onto *YText* instance.

get_array(*name: str*) → *YArray*

Returns
A *YArray* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned.

If there was an instance with this name, but it was of different type, it will be projected onto *YArray* instance.

get_map(*name: str*) → *YMap*

Returns
A *YMap* shared data type, that's accessible for subsequent accesses using given *name*.

If there was no instance with this name before, it will be created and then returned.

If there was an instance with this name, but it was of different type, it will be projected onto *YMap* instance.

commit()

Triggers a post-update series of operations without `free`ing the transaction. This includes compaction and optimization of internal representation of updates, triggering events etc. Ypy transactions are auto-committed when they are `free`d.

state_vector_v1() → EncodedStateVector

Encodes a state vector of a given transaction document into its binary representation using lib0 v1 encoding. State vector is a compact representation of updates performed on a given document and can be used by *encode_state_as_update* on remote peer to generate a delta update payload to synchronize changes between peers.

Example:

```
from y_py import YDoc

# document on machine A
local_doc = YDoc()
local_txn = local_doc.begin_transaction()

# document on machine B
remote_doc = YDoc()
remote_txn = local_doc.begin_transaction()

try:
```

(continues on next page)

(continued from previous page)

```
    local_sv = local_txn.state_vector_v1()
    remote_delta = remote_txn.diff_v1(local_sv)
    local_txn.apply_v1(remote_delta)
finally:
    del local_txn
    del remote_txn
```

diff_v1(*vector: Optional[EncodedStateVector] = None*) → YDocUpdate

Encodes all updates that have happened since a given version *vector* into a compact delta representation using lib0 v1 encoding. If *vector* parameter has not been provided, generated delta payload will contain all changes of a current Ypy document, working effectively as its state snapshot.

Example:

```
from y_py import YDoc

# document on machine A
local_doc = YDoc()
local_txn = local_doc.begin_transaction()

# document on machine B
remote_doc = YDoc()
remote_txn = local_doc.begin_transaction()

try:
    local_sv = local_txn.state_vector_v1()
    remote_delta = remote_txn.diff_v1(local_sv)
    local_txn.apply_v1(remote_delta)
finally:
    del local_txn
    del remote_txn
```

apply_v1(*diff: YDocUpdate*)

Applies delta update generated by the remote document replica to a current transaction's document. This method assumes that a payload maintains lib0 v1 encoding format.

Example:

```
from y_py import YDoc

# document on machine A
local_doc = YDoc()
local_txn = local_doc.begin_transaction()

# document on machine B
remote_doc = YDoc()
remote_txn = local_doc.begin_transaction()

try:
    local_sv = local_txn.state_vector_v1()
    remote_delta = remote_txn.diff_v1(local_sv)
    local_txn.apply_v1(remote_delta)
finally:
```

(continues on next page)

(continued from previous page)

```
    del local_txn
    del remote_txn
```

__enter__() → *YTransaction*

__exit__() → bool

class y_py.**YText**(*init: str = ''*)

A shared data type used for collaborative text editing. It enables multiple users to add and remove chunks of text in efficient manner. This type is internally represented as able double-linked list of text chunks - an optimization occurs during *YTransaction.commit*, which allows to squash multiple consecutively inserted characters together as a single chunk of text even between transaction boundaries in order to preserve more efficient memory model.

YText structure internally uses UTF-8 encoding and its length is described in a number of bytes rather than individual characters (a single UTF-8 code point can consist of many bytes).

Like all Yrs shared data types, *YText* is resistant to the problem of interleaving (situation when characters inserted one after another may interleave with other peers concurrent inserts after merging all updates together). In case of Yrs conflict resolution is solved by using unique document id to determine correct and consistent ordering.

prelim: bool

True if this element has not been integrated into a YDoc.

__str__() → str

Returns
The underlying shared string stored in this data type.

__repr__() → str

Returns
The string representation wrapped in 'YText()'

__len__() → int

Returns
The length of an underlying string stored in this *YText* instance, understood as a number of UTF-8 encoded bytes.

to_json() → str

Returns
The underlying shared string stored in this data type.

insert(*txn:* YTransaction, *index: int*, *chunk: str*, *attributes: Dict[str, Any] = {}*)

Inserts a string of text into the *YText* instance starting at a given *index*. Attributes are optional style modifiers (*{"bold": True}*) that can be attached to the inserted string. Attributes are only supported for a *YText* instance which already has been integrated into document store.

insert_embed(*txn:* YTransaction, *index: int*, *embed: Any*, *attributes: Dict[str, Any] = {}*)

Inserts embedded content into the YText at the provided index. Attributes are user-defined metadata associated with the embedded content. Attributes are only supported for a *YText* instance which already has been integrated into document store.

format(*txn:* YTransaction, *index: int*, *length: int*, *attributes: Dict[str, Any]*)

Wraps an existing piece of text within a range described by *index-length* parameters with formatting blocks containing provided *attributes* metadata. This method only works for *YText* instances that already have been integrated into document store

extend(*txn:* YTransaction, *chunk: str*)

Appends a given *chunk* of text at the end of current *YText* instance.

delete(*txn:* YTransaction, *index: int*)

Deletes the character at the specified *index*.

delete_range(*txn:* YTransaction, *index: int*, *length: int*)

Deletes a specified range of of characters, starting at a given *index*. Both *index* and *length* are counted in terms of a number of UTF-8 character bytes.

observe(*f: Callable[[*YTextEvent*]]*) → *SubscriptionId*

Assigns a callback function to listen to YText updates.

Parameters
f – Callback function that runs when the text object receives an update.

Returns
A reference to the callback subscription.

observe_deep(*f: Callable[[List[Event]]]*) → *SubscriptionId*

Assigns a callback function to listen to the updates of the YText instance and those of its nested attributes. Currently, this listens to the same events as YText.observe, but in the future this will also listen to the events of embedded values.

Parameters
f – Callback function that runs when the text object or its nested attributes receive an update.

Returns
A reference to the callback subscription.

unobserve(*subscription_id:* SubscriptionId)

Cancels the observer callback associated with the *subscripton_id*.

Parameters
subscription_id – reference to a subscription provided by the *observe* method.

class y_py.**YTextEvent**

Communicates updates that occurred during a transaction for an instance of *YText*. The *target* references the *YText* element that receives the update. The *delta* is a list of updates applied by the transaction.

target: *YText*

delta: **List[YTextDelta]**

path() → List[Union[int, str]]

Returns
Array of keys and indexes creating a path from root type down to current instance of shared type (accessible via *target* getter).

y_py.**YTextDelta**

class y_py.**YTextChangeInsert**

Bases: TypedDict

insert: **str**

attributes: **Optional[Any]**

class y_py.**YTextChangeDelete**

Bases: TypedDict

delete: int

class y_py.**YTextChangeRetain**

Bases: TypedDict

retain: int

attributes: Optional[Any]

class y_py.**YArray**

prelim: bool

True if this element has not been integrated into a YDoc.

__len__() → int

Returns
Number of elements in the *YArray*

__str__() → str

Returns
The string representation of YArray

__repr__() → str

Returns
The string representation of YArray wrapped in *YArray()*

to_json() → str

Converts an underlying contents of this *YArray* instance into their JSON representation.

insert(*txn:* YTransaction, *index: int*, *item: Any*)

Inserts an item at the provided index in the *YArray*.

insert_range(*txn:* YTransaction, *index: int*, *items: Iterable*)

Inserts a given range of *items* into this *YArray* instance, starting at given *index*.

append(*txn:* YTransaction, *item: Any*)

Adds a single item to the end of the *YArray*

extend(*txn:* YTransaction, *items: Iterable*)

Appends a sequence of *items* at the end of this *YArray* instance.

delete(*txn:* YTransaction, *index: int*)

Deletes a single item from the array

Parameters

- **txn** – The transaction where the array is being modified.
- **index** – The index of the element to be deleted.

delete_range(*txn:* YTransaction, *index: int*, *length: int*)

Deletes a range of items of given *length* from current *YArray* instance, starting from given *index*.

move_to(*txn: YTransaction*, *source: int*, *target: int*)

Moves a single item found at *source* index into *target* index position.

Parameters

- **txn** – The transaction where the array is being modified.
- **source** – The index of the element to be moved.
- **target** – The new position of the element.

move_range_to(*txn: YTransaction*, *start: int*, *end: int*, *target: int*)

Moves all elements found within *start*..`end` indexes range (both side inclusive) into new position pointed by *target* index. All elements inserted concurrently by other peers inside of moved range will be moved as well after synchronization (although it make take more than one sync roundtrip to achieve convergence).

Parameters

- **txn** – The transaction where the array is being modified.
- **start** – The index of the first element of the range (inclusive).
- **end** – The index of the last element of the range (inclusive).
- **target** – The new position of the element.

Example: ``` import y_py as Y doc = Y.Doc(); array = doc.get_array('array')

with doc.begin_transaction() as t:
array.insert_range(t, 0, [1,2,3,4]);

// move elements 2 and 3 after the 4 with doc.begin_transaction() as t:

array.move_range_to(t, 1, 2, 4);

```

**__getitem__**(*index: Union[int, slice]*) → Any

**Returns**
The element stored under given *index* or a new list of elements from the slice range.

**__iter__**() → Iterator

**Returns**
An iterator that can be used to traverse over the values stored withing this instance of *YArray*.

Example:

```
from y_py import YDoc

# document on machine A
doc = YDoc()
array = doc.get_array('name')

for item in array:
    print(item)
```

**observe**(*f: Callable[[YArrayEvent]]*) → *SubscriptionId*

Assigns a callback function to listen to YArray updates.

**Parameters**
**f** – Callback function that runs when the array object receives an update.
```

Returns
An identifier associated with the callback subscription.

observe_deep(*f: Callable[[List[Event]]]*) → *SubscriptionId*

Assigns a callback function to listen to the aggregated updates of the YArray and its child elements.

Parameters
f – Callback function that runs when the array object or components receive an update.

Returns
An identifier associated with the callback subscription.

unobserve(*subscription_id:* SubscriptionId)

Cancels the observer callback associated with the *subscripton_id*.

Parameters
subscription_id – reference to a subscription provided by the *observe* method.

y_py.**YArrayObserver**

class y_py.**YArrayEvent**

Communicates updates that occurred during a transaction for an instance of *YArray*. The *target* references the *YArray* element that receives the update. The *delta* is a list of updates applied by the transaction.

target: *YArray*

delta: **List[ArrayDelta]**

path() → List[Union[int, str]]

Returns
Array of keys and indexes creating a path from root type down to current instance of shared type (accessible via *target* getter).

y_py.**ArrayDelta**

A modification to a YArray during a transaction.

class y_py.**ArrayChangeInsert**

Bases: TypedDict

Update message that elements were inserted in a YArray.

insert: **List[Any]**

class y_py.**ArrayChangeDelete**

Update message that elements were deleted in a YArray.

delete: **int**

class y_py.**ArrayChangeRetain**

Update message that elements were left unmodified in a YArray.

retain: **int**

class y_py.**YMap**

prelim: **bool**

True if this element has not been integrated into a YDoc.

__len__() → int

Returns
The number of entries stored within this instance of *YMap*.

__str__() → str

Returns
The string representation of the *YMap*.

__dict__() → dict

Returns
Contents of the *YMap* inside a Python dictionary.

__repr__() → str

Returns
The string representation of the *YMap* wrapped in 'YMap()'

to_json() → str

Converts contents of this *YMap* instance into a JSON representation.

set(*txn:* YTransaction, *key: str*, *value: Any*)

Sets a given *key-value* entry within this instance of *YMap*. If another entry was already stored under given *key*, it will be overridden with new *value*.

update(*txn:* YTransaction, *items: Union[Iterable[Tuple[str, Any]], Dict[str, Any]]*)

Updates *YMap* with the contents of items.

Parameters

- **txn** – A transaction to perform the insertion updates.
- **items** – An iterable object that produces key value tuples to insert into the YMap

pop(*txn:* YTransaction, *key: str*, *fallback: Optional[Any] = None*) → Any

Removes an entry identified by a given *key* from this instance of *YMap*, if such exists. Throws a KeyError if the key does not exist and fallback value is not provided.

Parameters

- **txn** – The current transaction from a YDoc.
- **key** – Identifier of the requested item.
- **fallback** – Returns this value if the key doesn't exist in the YMap

Returns
The item at the key.

get(*key: str*, *fallback: Any*) → Any | None

Parameters

- **key** – The identifier for the requested data.
- **fallback** – If the key doesn't exist in the map, this fallback value will be returned.

Returns
Requested data or the provided fallback value.

__getitem__(*key: str*) → Any

Parameters
key – The identifier for the requested data.

Returns
Value of an entry stored under given *key* within this instance of *YMap*. Will throw a *KeyError* if the provided key is unassigned.

__iter__() → Iterator[str]

Returns
An iterator that traverses all keys of the *YMap* in an unspecified order.

items() → *YMapItemsView*

Returns
A view that can be used to iterate over all entries stored within this instance of *YMap*. Order of entry is not specified.

Example:

```
from y_py import YDoc

# document on machine A
doc = YDoc()
map = doc.get_map('name')
with doc.begin_transaction() as txn:
    map.set(txn, 'key1', 'value1')
    map.set(txn, 'key2', true)
for (key, value) in map.items()):
    print(key, value)
```

keys() → *YMapKeysView*

Returns
A view of all key identifiers in the YMap. The order of keys is not stable.

values() → *YMapValuesView*

Returns
A view of all values in the YMap. The order of values is not stable.

observe(*f: Callable[[*YMapEvent*]]*) → *SubscriptionId*

Assigns a callback function to listen to YMap updates.

Parameters
f – Callback function that runs when the map object receives an update.

Returns
A reference to the callback subscription. Delete this observer in order to erase the associated callback function.

observe_deep(*f: Callable[[List[Event]]]*) → *SubscriptionId*

Assigns a callback function to listen to YMap and child element updates.

Parameters
f – Callback function that runs when the map object or any of its tracked elements receive an update.

Returns
A reference to the callback subscription. Delete this observer in order to erase the associated callback function.

unobserve(*subscription_id:* SubscriptionId)

Cancels the observer callback associated with the *subscripton_id*.

Parameters
subscription_id – reference to a subscription provided by the *observe* method.

class y_py.**YMapItemsView**

Tracks key/values inside a YMap. Similar functionality to dict_items for a Python dict

__iter__() → Iterator[Tuple[str, Any]]

Produces key value tuples of elements inside the view

__contains__() → bool

Checks membership of kv tuples in the view

__len__() → int

Checks number of items in the view.

class y_py.**YMapKeysView**

Tracks key identifiers inside of a YMap

__iter__() → Iterator[str]

Produces keys of the view

__contains__() → bool

Checks membership of keys in the view

__len__() → int

Checks number of keys in the view.

class y_py.**YMapValuesView**

Tracks values inside of a YMap

__iter__() → Iterator[Any]

Produces values of the view

__contains__() → bool

Checks membership of values in the view

__len__() → int

Checks number of values in the view.

class y_py.**YMapEvent**

Communicates updates that occurred during a transaction for an instance of *YMap*. The *target* references the *YMap* element that receives the update. The *delta* is a list of updates applied by the transaction. The *keys* are a list of changed values for a specific key.

target: *YMap*

The element modified during this event.

keys: **Dict[str,** *YMapEventKeyChange***]**

A list of modifications to the YMap by key. Includes the type of modification along with the before and after state.

path() → List[Union[int, str]]

Returns
Path to this element from the root if this YMap is nested inside another data structure.

class y_py.**YMapEventKeyChange**

Bases: TypedDict

action: Literal[add, update, delete]

oldValue: Optional[Any]

newValue: Optional[Any]

y_py.**YXmlAttributes**

Generates a sequence of key/value properties for an XML Element

y_py.**Xml**

y_py.**YXmlTreeWalker**

Visits elements in an Xml tree

y_py.**EntryChange**

class y_py.**YXmlElementEvent**

target: *YXmlElement*

keys: Dict[str, EntryChange]

delta: List[Dict]

path() → List[Union[int, str]]

Returns a current shared type instance, that current event changes refer to.

class y_py.**YXmlElement**

XML element data type. It represents an XML node, which can contain key-value attributes (interpreted as strings) as well as other nested XML elements or rich text (represented by *YXmlText* type).

In terms of conflict resolution, *YXmlElement* uses following rules:

- Attribute updates use logical last-write-wins principle, meaning the past updates are automatically overridden and discarded by newer ones, while concurrent updates made by different peers are resolved into a single value using document id seniority to establish an order.
- Child node insertion uses sequencing rules from other Yrs collections - elements are inserted using interleave-resistant algorithm, where order of concurrent inserts at the same index is established using peer's document id seniority.

name: str

first_child: Optional[Xml]

next_sibling: Optional[Xml]

prev_sibling: Optional[Xml]

parent: Optional[*YXmlElement*]

__len__() → int

Returns a number of child XML nodes stored within this *YXMlElement* instance.

insert_xml_element(*txn:* YTransaction, *index: int*, *name: str*) → *YXmlElement*

Inserts a new instance of *YXmlElement* as a child of this XML node and returns it.

insert_xml_text(*txn:* YTransaction, *index: int*) → *YXmlText*

Inserts a new instance of *YXmlText* as a child of this XML node and returns it.

delete(*txn:* YTransaction, *index: int*, *length: int*)

Removes a range of children XML nodes from this *YXmlElement* instance, starting at given *index*.

push_xml_element(*txn:* YTransaction, *name: str*) → *YXmlElement*

Appends a new instance of *YXmlElement* as the last child of this XML node and returns it.

push_xml_text(*txn:* YTransaction) → *YXmlText*

Appends a new instance of *YXmlText* as the last child of this XML node and returns it.

__str__() → str

Returns
A string representation of this XML node.

__repr__() → str

Returns
A string representation wrapped in YXmlElement

set_attribute(*txn:* YTransaction, *name: str*, *value: str*)

Sets a *name* and *value* as new attribute for this XML node. If an attribute with the same *name* already existed on that node, its value with be overridden with a provided one.

get_attribute(*name: str*) → Optional[str]

Returns a value of an attribute given its *name*. If no attribute with such name existed, *null* will be returned.

remove_attribute(*txn:* YTransaction, *name: str*)

Removes an attribute from this XML node, given its *name*.

attributes() → YXmlAttributes

Returns an iterator that enables to traverse over all attributes of this XML node in unspecified order.

tree_walker() → YXmlTreeWalker

Returns an iterator that enables a deep traversal of this XML node - starting from first child over this XML node successors using depth-first strategy.

observe(*f: Callable[[*YXmlElementEvent*]]*) → *SubscriptionId*

Subscribes to all operations happening over this instance of *YXmlElement*. All changes are batched and eventually triggered during transaction commit phase.

Parameters
f – A callback function that receives update events.

Returns
A *SubscriptionId* that can be used to cancel the observer callback.

observe_deep(*f: Callable[[List[Event]]]*) → *SubscriptionId*

Subscribes to all operations happening over this instance of *YXmlElement* and its children. All changes are batched and eventually triggered during transaction commit phase.

Parameters
f – A callback function that receives update events from the Xml element and its children.

Returns
A *SubscriptionId* that can be used to cancel the observer callback.

unobserve(*subscription_id:* SubscriptionId)

Cancels the observer callback associated with the *subscripton_id*.

Parameters
subscription_id – reference to a subscription provided by the *observe* method.

class y_py.**YXmlText**

next_sibling: Optional[Xml]

prev_sibling: Optional[Xml]

parent: Optional[*YXmlElement*]

__len__()

Returns
The length of an underlying string stored in this *YXmlText* instance, understood as a number of UTF-8 encoded bytes.

insert(*txn:* YTransaction, *index: int*, *chunk: str*)

Inserts a given *chunk* of text into this *YXmlText* instance, starting at a given *index*.

push(*txn:* YTransaction, *chunk: str*)

Appends a given *chunk* of text at the end of *YXmlText* instance.

delete(*txn:* YTransaction, *index: int*, *length: int*)

Deletes a specified range of of characters, starting at a given *index*. Both *index* and *length* are counted in terms of a number of UTF-8 character bytes.

__str__() → str

Returns
The underlying string stored in this *YXmlText* instance.

__repr__() → str

Returns
The string representation wrapped in 'YXmlText()'

set_attribute(*txn:* YTransaction, *name: str*, *value: str*)

Sets a *name* and *value* as new attribute for this XML node. If an attribute with the same *name* already existed on that node, its value with be overridden with a provided one.

get_attribute(*name: str*) → Optional[str]

Returns
A value of an attribute given its *name*. If no attribute with such name existed,

None will be returned.

remove_attribute(*txn:* YTransaction, *name: str*)

Removes an attribute from this XML node, given its *name*.

attributes() → YXmlAttributes

Returns
An iterator that enables to traverse over all attributes of this XML node in

unspecified order.

observe(*f: Callable[[*YXmlTextEvent*]]*) → *SubscriptionId*

Subscribes to all operations happening over this instance of *YXmlText*. All changes are batched and eventually triggered during transaction commit phase.

Parameters

- **f** – A callback function that receives update events.
- **deep** – Determines whether observer is triggered by changes to elements in the YXmlText.

Returns
A *SubscriptionId* that can be used to cancel the observer callback.

observe_deep(*f: Callable[[List[Event]]]*) → *SubscriptionId*

Subscribes to all operations happening over this instance of *YXmlText* and its children. All changes are batched and eventually triggered during transaction commit phase.

Parameters

- **f** – A callback function that receives update events of this element and its descendants.
- **deep** – Determines whether observer is triggered by changes to elements in the YXmlText.

Returns
A *SubscriptionId* that can be used to cancel the observer callback.

unobserve(*subscription_id:* SubscriptionId)

Cancels the observer callback associated with the *subscripton_id*.

Parameters
subscription_id – reference to a subscription provided by the *observe* method.

class y_py.**YXmlTextEvent**

target: *YXmlText*

keys: List[EntryChange]

delta: List[YTextDelta]

path() → List[Union[int, str]]

Returns a current shared type instance, that current event changes refer to.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

y

INDEX

Symbols

A

B

C

D

E

F

G

I

K

M

N

O

P

R

S

T

U

V

X

Y