py-ssz Documentation

Release 0.5.0

The Ethereum Foundation

Apr 01, 2024

CONTENTS

Python implementation of the Simple Serialization encoding and decoding

CHAPTER

ONE

CONTENTS

1.1 ssz package

1.1.1 Subpackages

ssz.cache package

Submodules

ssz.cache.cache module

class ssz.cache.cache.**SSZCache**(*cache_size: int = 1024*)

Bases: MutableMapping

property cache_size: int

clear() → None. Remove all items from D.

ssz.cache.utils module

ssz.cache.utils.**get_base_key**(*sedes:* BaseSedes*[Any, Any]*, *value: Any*) → bytes

ssz.cache.utils.**get_key**(*sedes*, *value: Any*) → str

ssz.cache.utils.**get_merkle_leaves_with_cache**(*value: Any*, *element_sedes:* BaseSedes*[Any, Any]*, *cache: CacheObj*) → Iterable[Hash32]

Generate the merkle leaves for every element in *value*, from the cache.

NOTE: cache will be mutated when any new merkle leaves are generated.

ssz.cache.utils.**get_merkle_leaves_without_cache**(*value: Any*, *element_sedes:* BaseSedes*[Any, Any]*) → Iterable[Hash32]

Module contents

ssz.sedes package

Submodules

ssz.sedes.base module

class ssz.sedes.base.**BaseBitfieldCompositeSedes**(**args*, ***kwds*)

Bases: *BaseSedes*[TSerializable, TDeserialized]

class ssz.sedes.base.**BaseProperCompositeSedes**(**args*, ***kwds*)

Bases: *BaseSedes*[TSerializable, TDeserialized]

abstract property chunk_count: int | None

abstract property element_size_in_tree: int

abstract get_element_sedes(*index: int*) → *BaseSedes*

abstract property is_packing: bool

abstract serialize_element_for_tree(*index: int*, *element: TSerializable*) → bytes

class ssz.sedes.base.**BaseSedes**(**args*, ***kwds*)

Bases: ABC, Generic[TSerializable, TDeserialized]

abstract deserialize(*data: bytes*) → TDeserialized

abstract get_fixed_size() → int

abstract get_hash_tree_root(*value: TSerializable*) → Hash32

abstract get_hash_tree_root_and_leaves(*value: TSerializable*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

abstract get_key(*value: Any*) → str

abstract get_sedes_id() → str

abstract property is_fixed_sized: bool

abstract serialize(*value: TSerializable*) → bytes

ssz.sedes.basic module

class ssz.sedes.basic.**BasicSedes**(*size: int*)

Bases: *BaseSedes*[TSerializable, TDeserialized]

get_fixed_size()

get_hash_tree_root(*value: TSerializable*) → Hash32

get_hash_tree_root_and_leaves(*value: TSerializable*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

get_key(*value: Any*) → str

is_fixed_sized = True

class ssz.sedes.basic.**BitfieldCompositeSedes**(**args*, ***kwds*)

Bases: *BaseBitfieldCompositeSedes*[TSerializable, TDeserialized]

get_key(*value: Any*) → str

class ssz.sedes.basic.**HomogeneousProperCompositeSedes**(**args*, ***kwds*)

Bases: *ProperCompositeSedes*[TSerializable, TDeserialized]

property chunk_count: int

get_sedes_id() → str

property is_packing: bool

class ssz.sedes.basic.**ProperCompositeSedes**(**args*, ***kwds*)

Bases: *BaseProperCompositeSedes*[TSerializable, TDeserialized]

deserialize(*data: bytes*) → TDeserialized

property element_size_in_tree: int

get_key(*value: Any*) → str

serialize(*value: TSerializable*) → bytes

serialize_element_for_tree(*index: int*, *element: TSerializable*) → bytes

ssz.sedes.bitlist module

class ssz.sedes.bitlist.**Bitlist**(*max_bit_count: int*)

Bases: *BitfieldCompositeSedes*[Union[bytes, bytearray], bytes]

property chunk_count: int

deserialize(*data: bytes*) → Tuple[bool, ...]

get_fixed_size()

get_hash_tree_root(*value: Sequence[bool]*) → bytes

get_hash_tree_root_and_leaves(*value: Sequence[bool]*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

get_sedes_id() → str

is_fixed_sized = False

serialize(*value: Sequence[bool]*) → bytes

ssz.sedes.bitlist.**get_bitlist_len**(*x: int*) → int

ssz.sedes.bitvector module

class ssz.sedes.bitvector.**Bitvector**(*bit_count: int*)

Bases: *BitfieldCompositeSedes*[Union[bytes, bytearray], bytes]

property chunk_count: int

deserialize(*data: bytes*) → bytes

get_fixed_size()

get_hash_tree_root(*value: Sequence[bool]*) → bytes

get_hash_tree_root_and_leaves(*value: Sequence[bool]*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

get_sedes_id() → str

is_fixed_sized = True

serialize(*value: Sequence[bool]*) → bytes

ssz.sedes.boolean module

class ssz.sedes.boolean.**Bit**

Bases: *Boolean*

class ssz.sedes.boolean.**Boolean**

Bases: *BasicSedes*[bool, bool]

deserialize(*data: bytes*) → bool

get_sedes_id() → str

serialize(*value: bool*) → bytes

ssz.sedes.byte module

class ssz.sedes.byte.**Byte**

Bases: *BasicSedes*[bytes, bytes]

deserialize(*data: bytes*) → bytes

get_sedes_id() → str

serialize(*value: bytes*) → bytes

size = 1

ssz.sedes.byte_list module

class ssz.sedes.byte_list.**ByteList**(*max_length: int*)

Bases: *List*[Union[bytes, bytearray], bytes]

Equivalent to *List(byte, size)* but more convenient & efficient.

When encoding a series of bytes, List(byte, ...) requires an awkward input shaped like: `(b'A', b'B', b'C')`. *ByteList* accepts a simple bytes object like `b'ABC'` for encoding.

deserialize(*data: bytes*) → bytes

get_hash_tree_root(*value: bytes*) → bytes

get_sedes_id() → str

serialize(*value: bytes | bytearray*) → bytes

serialize_element_for_tree(*index: int*, *byte_value: int*) → bytes

ssz.sedes.byte_vector module

class ssz.sedes.byte_vector.**ByteVector**(*size: int*)

Bases: *Vector*[Union[bytes, bytearray], bytes]

Equivalent to *Vector(byte, size)* but more efficient.

deserialize(*data: bytes*) → bytes

get_hash_tree_root(*value: bytes*) → bytes

get_hash_tree_root_and_leaves(*value: bytes*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

get_sedes_id() → str

serialize(*value: bytes | bytearray*) → bytes

serialize_element_for_tree(*index: int*, *byte_value: int*) → bytes

ssz.sedes.container module

class ssz.sedes.container.**Container**(*field_sedes: Sequence[*BaseSedes*[Any, Any]]*)

Bases: *ProperCompositeSedes*[Sequence[Any], Tuple[Any, ...]]

property **chunk_count**: int

deserialize_fixed_size_parts(*stream: IO[bytes]*) → Iterable[Tuple[Tuple[Any], Tuple[int, *BaseSedes*[Any, Any]]]]

deserialize_variable_size_parts(*offset_pairs: Tuple[Tuple[int,* BaseSedes*[Any, Any]], ...]*, *stream: IO[bytes]*) → Iterable[Any]

get_element_sedes(*index: int*) → *BaseSedes*

get_fixed_size()

get_hash_tree_root(*value: Tuple[Any, ...]*) → bytes

get_hash_tree_root_and_leaves(*value: Tuple[Any, ...]*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

get_sedes_id() → str

property is_fixed_sized

property is_packing: bool

serialize(*value*) → bytes

ssz.sedes.list module

class ssz.sedes.list.**List**(*element_sedes:* BaseSedes*[Any, Any]*, *max_length: int*)

Bases: *HomogeneousProperCompositeSedes*[Sequence[TSerializable], Tuple[TDeserialized, ...]]

get_element_sedes(*index*) → *BaseSedes*[TSerializable, TDeserialized]

get_fixed_size()

get_hash_tree_root(*value: Iterable[TSerializable]*) → bytes

get_hash_tree_root_and_leaves(*value: TSerializable*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

is_fixed_sized = False

ssz.sedes.serializable module

class ssz.sedes.serializable.**BaseSerializable**(**args*, *cache=None*, ***kwargs*)

Bases: Sequence

as_dict()

cache = None

copy(**args*, ***kwargs*)

get_key() → bytes

classmethod get_sedes_id() → str

property hash_tree_root

reset_cache()

class ssz.sedes.serializable.**Meta**(*has_fields*, *fields*, *container_sedes*, *field_names*, *field_attrs*)

Bases: tuple

container_sedes: *Container* | None

Alias for field number 2

field_attrs: Tuple[str, ...] | None

Alias for field number 4

field_names: Tuple[str, ...] | None

Alias for field number 3

fields: Tuple[Tuple[str, *BaseSedes*], ...] | None
Alias for field number 1

has_fields: bool
Alias for field number 0

class ssz.sedes.serializable.**MetaSerializable**(*name*, *bases*, *namespace*)
Bases: ABCMeta

deserialize(*data: bytes*) → TSerializable

get_fixed_size()

get_hash_tree_root(*value: TSerializable*, *cache: bool = True*) → bytes

property is_fixed_sized

serialize(*value: TSerializable*) → bytes

class ssz.sedes.serializable.**Serializable**(**args*, *cache=None*, ***kwargs*)
Bases: *BaseSerializable*

The base class for serializable objects.

ssz.sedes.serializable.**make_immutable**(*value*)

ssz.sedes.serializable.**merge_args_to_kwargs**(*args*, *kwargs*, *arg_names*)

ssz.sedes.serializable.**merge_kwargs_to_args**(*args*, *kwargs*, *arg_names*)

ssz.sedes.serializable.**validate_args_and_kwargs**(*args*, *kwargs*, *arg_names*)

ssz.sedes.signed_serializable module

class ssz.sedes.signed_serializable.**MetaSignedSerializable**(*name*, *bases*, *namespace*)
Bases: *MetaSerializable*

class ssz.sedes.signed_serializable.**SignedMeta**(*has_fields*, *fields*, *container_sedes*, *signed_container_sedes*, *field_names*, *field_attrs*)
Bases: tuple

container_sedes: *Container* | None
Alias for field number 2

field_attrs: Tuple[str, ...] | None
Alias for field number 5

field_names: Tuple[str, ...] | None
Alias for field number 4

fields: Tuple[Tuple[str, *BaseSedes*]] | None
Alias for field number 1

has_fields: bool
Alias for field number 0

signed_container_sedes: *Container* | None
Alias for field number 3

class ssz.sedes.signed_serializable.**SignedSerializable**(**args*, *cache=None*, ***kwargs*)

Bases: *BaseSerializable*

property signing_root

ssz.sedes.uint module

class ssz.sedes.uint.**UInt**(*num_bits: int*)

Bases: *BasicSedes*[int, int]

deserialize(*data: bytes*) → int

get_sedes_id() → str

serialize(*value: int*) → bytes

ssz.sedes.vector module

class ssz.sedes.vector.**Vector**(*element_sedes:* BaseSedes*[Any, Any]*, *length: int*)

Bases: *HomogeneousProperCompositeSedes*[Sequence[TSerializable], Tuple[TDeserialized, ...]]

get_element_sedes(*index*) → *BaseSedes*[TSerializable, TDeserialized]

get_fixed_size() → int

get_hash_tree_root(*value: Sequence[Any]*) → bytes

get_hash_tree_root_and_leaves(*value: Sequence[Any]*, *cache: CacheObj*) → Tuple[Hash32, CacheObj]

property is_fixed_sized: bool

property length: int

Module contents

ssz.sedes.**infer_sedes**(*value*)

Try to find a sedes objects suitable for a given Python object.

ssz.tools package

Submodules

ssz.tools.codec module

class ssz.tools.codec.**DefaultCodec**

Bases: object

static decode_bool(*value*, *sedes*) → bool

static decode_bytes(*value*, *sedes*) → bytes

static decode_integer(*value*, *sedes*) → int

static encode_bool(*value: bool*, *sedes*)

static encode_bytes(*value: bytes*, *sedes*)

static encode_integer(*value: int*, *sedes*)

ssz.tools.dump module

ssz.tools.dump.**dump**(*value*, *sedes=None*, *codec=<class 'ssz.tools.codec.DefaultCodec'>*)

ssz.tools.dump.**dump_bits**(*value*, *sedes*, *codec*)

ssz.tools.dump.**dump_boolean**(*value*, *sedes*, *codec*)

ssz.tools.dump.**dump_bytes**(*value*, *sedes*, *codec*)

ssz.tools.dump.**dump_container**(*value*, *sedes*, *codec*)

ssz.tools.dump.**dump_hashable_container**(*value*, *codec*)

ssz.tools.dump.**dump_hashable_sequence**(*value*, *codec*)

ssz.tools.dump.**dump_integer**(*value*, *sedes*, *codec*)

ssz.tools.dump.**dump_list**(*value*, *sedes*, *codec*)

ssz.tools.dump.**dump_serializable**(*value*, *codec*)

ssz.tools.dump.**dump_vector**(*value*, *sedes*, *codec*)

ssz.tools.dump.**to_formatted_dict**(*value*, *sedes=None*, *codec=<class 'ssz.tools.codec.DefaultCodec'>*)

ssz.tools.parse module

ssz.tools.parse.**from_formatted_dict**(*value*, *sedes*, *codec=<class 'ssz.tools.codec.DefaultCodec'>*)

ssz.tools.parse.**parse**(*value*, *sedes*, *codec=<class 'ssz.tools.codec.DefaultCodec'>*)

ssz.tools.parse.**parse_bits**(*value*, *sedes*, *codec*)

ssz.tools.parse.**parse_boolean**(*value*, *sedes*, *codec*)

ssz.tools.parse.**parse_bytes**(*value*, *sedes*, *codec*)

ssz.tools.parse.**parse_container**(*value*, *sedes*, *codec*)

ssz.tools.parse.**parse_hashable**(*value*, *hashable_cls*, *codec*)

ssz.tools.parse.**parse_integer**(*value*, *sedes*, *codec*)

ssz.tools.parse.**parse_list**(*value*, *sedes*, *codec*)

ssz.tools.parse.**parse_serializable**(*value*, *serializable_cls*, *codec*)

ssz.tools.parse.**parse_vector**(*value*, *sedes*, *codec*)

Module contents

1.1.2 Submodules

1.1.3 ssz.abc module

class ssz.abc.**HashableStructureAPI**(**args*, ***kwds*)

Bases: ABC, Generic[TElement]

abstract property chunks: PVector[Hash32]

abstract property elements: PVector[TElement]

abstract evolver() → *HashableStructureEvolverAPI*[TStructure, TElement]

abstract classmethod from_iterable_and_sedes(*iterable: Iterable[TElement]*, *sedes:* BaseProperCompositeSedes, *max_length: int* | *None*)

abstract property hash_tree: *HashTree*

abstract property hash_tree_root: Hash32

abstract mset(**args: int* | *TElement*) → TStructure

abstract property raw_root: Hash32

abstract set(*index: int*, *value: TElement*) → TStructure

abstract transform(**transformations*)

class ssz.abc.**HashableStructureEvolverAPI**(*hashable_structure: TStructure*)

Bases: ABC, Generic[TStructure, TElement]

abstract is_dirty() → bool

abstract persistent() → TStructure

abstract set(*index: int*, *element: TElement*) → None

class ssz.abc.**ResizableHashableStructureAPI**(**args*, ***kwds*)

Bases: *HashableStructureAPI*[TElement]

abstract append(*value: TElement*) → TStructure

abstract evolver() → *ResizableHashableStructureEvolverAPI*[TStructure, TElement]

abstract extend(*values: Iterable[TElement]*) → TStructure

class ssz.abc.**ResizableHashableStructureEvolverAPI**(*hashable_structure: TStructure*)

Bases: *HashableStructureEvolverAPI*[TStructure, TElement]

abstract append(*element: TElement*) → None

abstract extend(*iterable: Iterable[TElement]*) → None

1.1.4 ssz.codec module

ssz.codec.**decode**(*ssz*, *sedes*)

Decode a SSZ encoded object.

ssz.codec.**encode**(*value*, *sedes=None*)

Encode object in SSZ format. *sedes* needs to be explicitly mentioned for encode/decode of integers(as of now). *sedes* parameter could be given as a string or as the actual sedes object itself.

1.1.5 ssz.constants module

1.1.6 ssz.exceptions module

exception ssz.exceptions.**DeserializationError**

Bases: *SSZException*

Exception raised if deserialization fails.

exception ssz.exceptions.**SSZException**

Bases: Exception

Base class for exceptions raised by this package.

exception ssz.exceptions.**SerializationError**

Bases: *SSZException*

Exception raised if serialization fails.

1.1.7 ssz.hash module

ssz.hash.**hash_eth2**(*data: bytes*) → Hash32

Return SHA-256 hashed result. Note: it's a placeholder and we aim to migrate to a S[T/N]ARK-friendly hash function in a future Ethereum 2.0 deployment phase.

1.1.8 ssz.hash_tree module

class ssz.hash_tree.**HashTree**(*raw_hash_tree: PVector[PVector[Hash32]]*, *chunk_count: int | None = None*)

Bases: PVector[Hash32]

append(*value: Hash32*) → *HashTree*

property chunks: PVector[Hash32]

classmethod compute(*chunks: Iterable[Hash32]*, *chunk_count: int | None = None*) → *HashTree*

count(*value*) → integer -- return number of occurrences of value

delete(*index: int*, *stop: int | None = None*) → *HashTree*

evolver()

extend(*value: Iterable[Hash32]*) → *HashTree*

index(*value*[, *start*[, *stop*]]) → integer -- return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

mset(**args: int | Hash32*) → *HashTree*

remove(*value: Hash32*) → *HashTree*

property root: Hash32

set(*index: int*, *value: Hash32*) → *HashTree*

transform(**transformations*)

class ssz.hash_tree.**HashTreeEvolver**(*hash_tree:* HashTree)

Bases: object

append(*value: Hash32*) → None

delete(*index*, *stop=None*)

extend(*values: Iterable[Hash32]*) → None

is_dirty() → bool

persistent() → *HashTree*

remove(*value*)

set(*index: Integral*, *value: Hash32*) → None

ssz.hash_tree.**append_chunk_to_tree**(*hash_tree: PVector[PVector[Hash32]]*, *chunk: Hash32*) → PVector[PVector[Hash32]]

ssz.hash_tree.**compute_hash_tree**(*chunks: Iterable[Hash32]*, *chunk_count: int | None = None*) → PVector[PVector[Hash32]]

ssz.hash_tree.**generate_chunk_tree_padding**(*unpadded_chunk_tree: PVector[Hash32]*, *chunk_count: int | None*) → Generator[Hash32, None, None]

ssz.hash_tree.**generate_hash_tree_layers**(*chunks: PVector[Hash32]*) → Generator[PVector[Hash32], None, None]

ssz.hash_tree.**get_num_layers**(*num_chunks: int*, *chunk_count: int | None*) → int

ssz.hash_tree.**hash_layer**(*child_layer: PVector[Hash32]*, *layer_index: int*) → PVector[Hash32]

ssz.hash_tree.**pad_hash_tree**(*unpadded_chunk_tree: PVector[PVector[Hash32]]*, *chunk_count: int | None = None*) → PVector[PVector[Hash32]]

ssz.hash_tree.**recompute_hash_in_tree**(*hash_tree: PVector[PVector[Hash32]]*, *layer_index: int*, *hash_index: int*) → PVector[PVector[Hash32]]

ssz.hash_tree.**set_chunk_in_tree**(*hash_tree: PVector[PVector[Hash32]]*, *index: int*, *chunk: Hash32*) → PVector[PVector[Hash32]]

ssz.hash_tree.**validate_chunk_count**(*chunk_count: int | None*) → None

ssz.hash_tree.**validate_raw_hash_tree**(*raw_hash_tree: PVector[PVector[Hash32]]*, *chunk_count: int | None = None*) → None

1.1.9 ssz.hashable_container module

class ssz.hashable_container.**FieldDescriptor**(*name: str*)

Bases: object

Descriptor translating from __getattr__ to __getitem__ calls for a given attribute.

ssz.hashable_container.**GenericMetaHashableContainer**

alias of *MetaHashableContainer*

ssz.hashable_container.**GenericMetaSignedHashableContainer**

alias of *MetaSignedHashableContainer*

class ssz.hashable_container.**HashableContainer**(**args*, ***kwargs*)

Bases: *BaseHashableStructure*[TElement]

Base class for hashable containers.

classmethod create(***field_kwargs: Dict[str, Any]*)

evolver() → *HashableContainerEvolver*[TStructure, TElement]

property hash_tree_root: Hash32

normalize_item_index(*index: str | int*) → int

class ssz.hashable_container.**HashableContainerEvolver**(*hashable_structure: TStructure*)

Bases: *HashableStructureEvolver*[TStructure, TElement]

Base class for evolvers for hashable containers.

Subclasses (created dynamically by MetaHashableContainer when creating the corresponding HashableContainer) should add settable field descriptors for all fields.

class ssz.hashable_container.**Meta**(*fields*, *field_names*, *field_names_to_element_indices*, *container_sedes*, *evolver_class*)

Bases: tuple

container_sedes: *Container*

Alias for field number 3

evolver_class: Type[*HashableContainerEvolver*]

Alias for field number 4

field_names: Tuple[str, ...]

Alias for field number 1

field_names_to_element_indices: Dict[str, int]

Alias for field number 2

fields: Tuple[Tuple[str, *BaseSedes*], ...]

Alias for field number 0

classmethod from_fields(*fields: Tuple[Tuple[str,* BaseSedes*], ...]*, *container:* Container, *name: str*) → *Meta*

class ssz.hashable_container.**MetaHashableContainer**(*name: str*, *bases: Tuple[Type, ...]*, *namespace: Dict[str, Any]*)

Bases: ABCMeta

Metaclass which creates HashableContainers.

deserialize(*data*)

get_fixed_size()

get_hash_tree_root(*value*)

get_hash_tree_root_and_leaves(*value*, *cache*)

get_key(*value*)

get_sedes_id()

property is_fixed_sized

serialize(*value*)

class ssz.hashable_container.**MetaSignedHashableContainer**(*name*, *bases*, *namespace*)

Bases: *MetaHashableContainer*

class ssz.hashable_container.**SettableFieldDescriptor**(*name: str*)

Bases: *FieldDescriptor*

Settable variant of FieldDescriptor for evolver.

class ssz.hashable_container.**SignedHashableContainer**(**args*, ***kwargs*)

Bases: *HashableContainer*[TElement]

property signing_root: Hash32

ssz.hashable_container.**get_field_sedes_from_fields**(*fields: Sequence[Tuple[str,* BaseSedes*]]*) → Generator[*BaseSedes*, None, None]

ssz.hashable_container.**get_meta_from_bases**(*bases: Tuple[Type, ...]*) → *Meta* | None

Return the meta object defined by one of the given base classes.

Returns None if no base defines a meta object. Raises a TypeError if more than one do.

ssz.hashable_container.**hashablify_field_kwargs**(*field_kwargs: Dict[str, Any]*, *fields: Sequence[Tuple[str,* BaseSedes*]]*) → Generator[Tuple[str, Any], None, None]

ssz.hashable_container.**hashablify_value**(*value: Any*, *sedes:* BaseSedes) → Any

1.1.10 ssz.hashable_list module

class ssz.hashable_list.**HashableList**(*elements: PVector[TElement]*, *hash_tree:* HashTree, *sedes:* BaseProperCompositeSedes, *max_length: int* | *None* = *None*)

Bases: *BaseResizableHashableStructure*[TElement], Sequence[TElement]

classmethod from_iterable(*iterable: Iterable[TElement]*, *sedes:* List*[TElement, TElement]*)

property hash_tree_root: Hash32

1.1.11 ssz.hashable_structure module

class ssz.hashable_structure.**BaseHashableStructure**(*elements: PVector[TElement]*, *hash_tree:* HashTree, *sedes:* BaseProperCompositeSedes, *max_length: int | None = None*)

Bases: *HashableStructureAPI*[TElement]

property chunks: PVector[Hash32]

property elements: PVector[TElement]

evolver() → *HashableStructureEvolverAPI*[TStructure, TElement]

classmethod from_iterable_and_sedes(*iterable: Iterable[TElement]*, *sedes:* BaseProperCompositeSedes, *max_length: int | None = None*)

property hash_tree: *HashTree*

property max_length: int | None

mset(**args: int | TElement*) → TStructure

property raw_root: Hash32

property sedes: *BaseProperCompositeSedes*

set(*index: int*, *value: TElement*) → TStructure

transform(**transformations*)

class ssz.hashable_structure.**BaseResizableHashableStructure**(*elements: PVector[TElement]*, *hash_tree:* HashTree, *sedes:* BaseProperCompositeSedes, *max_length: int | None = None*)

Bases: *BaseHashableStructure*, *ResizableHashableStructureAPI*[TElement]

append(*value: TElement*) → TResizableStructure

evolver() → *ResizableHashableStructureEvolverAPI*[TResizableStructure, TElement]

extend(*values: Iterable[TElement]*) → TResizableStructure

class ssz.hashable_structure.**HashableStructureEvolver**(*hashable_structure: TStructure*)

Bases: *HashableStructureEvolverAPI*[TStructure, TElement]

is_dirty() → bool

persistent() → TStructure

set(*index: int*, *element: TElement*) → None

class ssz.hashable_structure.**ResizableHashableStructureEvolver**(*hashable_structure: TStructure*)

Bases: *HashableStructureEvolver*, *ResizableHashableStructureEvolverAPI*[TStructure, TElement]

append(*element: TElement*) → None

extend(*elements: Iterable[TElement]*) → None

ssz.hashable_structure.**get_appended_chunks**(*, *appended_elements: Sequence[bytes]*, *element_size: int*, *num_padding_elements: int*) → Generator[Hash32, None, None]

Get the sequence of appended chunks.

ssz.hashable_structure.**get_num_padding_elements**(*, *num_original_elements: int*, *num_original_chunks: int*, *element_size: int*) → int

Compute the number of elements that would still fit in the empty space of the last chunk.

ssz.hashable_structure.**get_updated_chunks**(*, *updated_elements: Dict[int, bytes]*, *appended_elements: Sequence[bytes]*, *original_chunks: Sequence[Hash32]*, *element_size: int*, *num_original_elements: int*, *num_padding_elements: int*) → Generator[Tuple[int, Hash32], None, None]

For an element changeset, compute the updates that have to be applied to the existing chunks.

The changeset is given as a dictionary of element indices to updated elements and a sequence of appended elements. Note that appended elements that do not affect existing chunks are ignored.

The pre-existing state is given by the sequence of original chunks and the number of elements represented by these chunks.

The return value is a dictionary mapping chunk indices to chunks.

ssz.hashable_structure.**update_element_in_chunk**(*original_chunk: Hash32*, *index: int*, *element: bytes*) → Hash32

Replace part of a chunk with a given element.

The chunk is interpreted as a concatenated sequence of equally sized elements. This function replaces the element given by its index in the chunk with the given data.

If the length of the element is zero or not a divisor of the chunk size, a *ValueError* is raised. If the index is out of range, an *IndexError* is raised.

```
>>> from ssz.hashable_structure import update_element_in_chunk
>>> update_element_in_chunk(b"aabbcc", 1, b"xx")
b'aaxxcc'
```

ssz.hashable_structure.**update_elements_in_chunk**(*original_chunk: Hash32*, *updated_elements: Dict[int, bytes]*) → Hash32

Update multiple elements in a chunk.

The set of updates is given by a dictionary mapping indices to elements. The items of the dictionary will be passed one by one to *update_element_in_chunk*.

1.1.12 ssz.hashable_vector module

class ssz.hashable_vector.**HashableVector**(*elements: PVector[TElement]*, *hash_tree:* HashTree, *sedes:* BaseProperCompositeSedes, *max_length: int | None = None*)

Bases: *BaseHashableStructure*[TElement], Sequence[TElement]

classmethod from_iterable(*iterable: Iterable[TElement]*, *sedes:* Vector*[TElement, TElement]*)

property hash_tree_root: Hash32

1.1.13 ssz.tree_hash module

ssz.tree_hash.**get_hash_tree_root**(*value: Any*, *sedes:* BaseSedes | *None* = *None*) → Hash32

1.1.14 ssz.typing module

1.1.15 ssz.utils module

ssz.utils.**decode_offset**(*data: bytes*) → int

ssz.utils.**encode_offset**(*offset: int*) → bytes

ssz.utils.**get_duplicates**(*values*)

ssz.utils.**get_items_per_chunk**(*item_size: int*) → int

ssz.utils.**get_next_power_of_two**(*value: int*) → int

ssz.utils.**get_serialized_bytearray**(*value: Sequence[bool]*, *bit_count: int*, *extra_byte: bool*) → bytearray

ssz.utils.**is_immutable_field_value**(*value: Any*) → bool

ssz.utils.**merkleize**(*chunks: Sequence[Hash32]*, *limit: int* | *None* = *None*) → Hash32

ssz.utils.**merkleize_with_cache**(*chunks: Sequence[Hash32]*, *cache: CacheObj*, *limit: int* | *None* = *None*) → Tuple[Hash32, CacheObj]

ssz.utils.**mix_in_length**(*root: Hash32*, *length: int*) → Hash32

ssz.utils.**pack**(*serialized_values: Sequence[bytes]*) → Tuple[Hash32, ...]

ssz.utils.**pack_bits**(*values: Sequence[bool]*) → Tuple[Hash32]

ssz.utils.**pack_bytes**(*byte_string: bytes*) → Tuple[bytes, ...]

ssz.utils.**pad_zeros**(*value: bytes*) → bytes

ssz.utils.**read_exact**(*num_bytes: int*, *stream: IO[bytes]*) → bytes

ssz.utils.**s_decode_offset**(*stream: IO[bytes]*) → int

ssz.utils.**to_chunks**(*packed_data: bytes*) → Tuple[bytes, ...]

1.1.16 Module contents

1.2 Release Notes

1.2.1 py-ssz v0.5.0 (2024-04-01)

Internal Changes - for py-ssz Contributors

- Add `python 3.12` support, add all doc type tests and night runs to CI, add blocklint to linting, turned off `yaml` tests in CI (#136)

1.2.2 py-ssz v0.4.0 (2023-12-07)

Breaking Changes

- Drop support for python 3.7 (#134)

Internal Changes - for py-ssz Contributors

- Add `build.os` config to readthedocs.yml (#133)
- Merge changes from the template, including use `pre-commit` for linting and change the name of the `master` branch to `main` (#134)

1.2.3 py-ssz v0.3.1 (2023-06-08)

Internal Changes - for py-ssz Contributors

- pull in most project template updates (#128)
- bump eth-utils requirement version to >=2 (#132)

1.2.4 py-ssz v0.3.0 (2022-08-19)

Breaking changes

- Dropping official support for Python 3.6 (although it still worked as of the last test run). (#125)

Features

- Add a *`ByteList`* sedes that is more convenient and more performant. With ByteList, the caller can decode a `bytes` object, rather than passing in a list of single-byte elements. (#118)

Bugfixes

- Reject empty bytes at the end of a bitlist as invalid (#109)
- Reject vectors and bitvectors of length 0 as invalid, as defined in the spec. (#111)
- Enforce that vector types must have a maximum length of 1 or more, and lists may have a 0 max length (#116)

Improved Documentation

- Sort release notes with most recent on top (#124)

Internal Changes - for py-ssz Contributors

- Upgrade black to a stable version, and pass newest style checks (#120)
- Use the latest project template, which gives many developer-focused benefits: in making release notes, releasing new versions, etc. (#121)
- Miscellaneous changes (#124):
  - Run black autoformat, as part of `make lint-roll`
  - Added some tests to check length validation of *`ByteList`* and *`ByteVector`*
  - When generating website docs from docstrings, skip tests

1.2.5 v0.2.4

Released 2020-03-24

- Update *pyrsistent* dependency.

1.2.6 v0.1.0-alpha.8

Released 2018-05-05

- Less strict class relationship requirement for equality of serializables - #71

1.2.7 v0.1.0-alpha.7

Released 2018-05-02

- Fix equality of serializable objects - #64
- Add helpers to convert objects to and from a human readable representation - #66
- Cache hash tree root of serializable objects - #68

1.2.8 v0.1.0-alpha.6

Released 2018-04-23

No changes

1.2.9 v0.1.0-alpha.5

Released 2018-04-23

- Slight change in serializable inheritance rules - #57
- Add root property to serializable - #57
- Add SignedSerializable base class with signing-root property - #57

1.2.10 v0.1.0-alpha.4

Released 2018-04-09

- Fix bug in serializable class - #56

1.2.11 v0.1.0-alpha.3

Released 2018-04-04

- Implement spec version 0.5.0

1.2.12 v0.1.0-alpha.2

Released 2018-02-05

- Add zero padding to tree hash - #35

1.2.13 v0.1.0-alpha.1

Released 2018-02-05

- Implements January pre-release spec

1.2.14 v0.1.0-alpha.0

- Launched repository, claimed names for pip, RTD, github, etc

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

S

INDEX

E

F

G

H

I

L

M

N

P

R

S

T

U

V