class **HeteroData** (**_mapping**: Optional[Dict[str, Any]] = **None**, ****kwargs**) [source]

Bases: `BaseData` , `FeatureStore` , `GraphStore`

A data object describing a heterogeneous graph, holding multiple node and/or edge types in disjunct storage objects. Storage objects can hold either node-level, link-level or graph-level attributes. In general, `HeteroData` tries to mimic the behavior of a regular **nested** Python dictionary. In addition, it provides useful functionality for analyzing graph structures, and provides basic PyTorch tensor functionalities.

```
from torch_geometric.data import HeteroData

data = HeteroData()

# Create two node types "paper" and "author" holding a feature matrix:
data['paper'].x = torch.randn(num_papers, num_paper_features)
data['author'].x = torch.randn(num_authors, num_authors_features)

# Create an edge type "(author, writes, paper)" and building the
# graph connectivity:
data['author', 'writes', 'paper'].edge_index = ...  # [2, num_edges]

data['paper'].num_nodes
>>> 23

data['author', 'writes', 'paper'].num_edges
>>> 52

# PyTorch tensor functionality:
data = data.pin_memory()
data = data.to('cuda:0', non_blocking=True)
```

Note that there exists multiple ways to create a heterogeneous graph data, *e.g.*:

- To initialize a node of type `"paper"` holding a node feature matrix `x_paper` named `x` :

```
from torch_geometric.data import HeteroData

# (1) Assign attributes after initialization,
data = HeteroData()
data['paper'].x = x_paper

# or (2) pass them as keyword arguments during initialization,
data = HeteroData(paper={ 'x': x_paper })

# or (3) pass them as dictionaries during initialization,
data = HeteroData({'paper': { 'x': x_paper }})
```

- To initialize an edge from source node type `"author"` to destination node type `"paper"` with relation type `"writes"` holding a graph connectivity matrix `edge_index_author_paper` named `edge_index`:

```
# (1) Assign attributes after initialization,
data = HeteroData()
data['author', 'writes', 'paper'].edge_index = edge_index_author_paper

# or (2) pass them as keyword arguments during initialization,
data = HeteroData(author__writes__paper={
    'edge_index': edge_index_author_paper
})

# or (3) pass them as dictionaries during initialization,
data = HeteroData({
    ('author', 'writes', 'paper'):
    { 'edge_index': edge_index_author_paper }
})
```

classmethod **from_dict** (**mapping**: Dict[str, Any]) → Self [source]

Creates a `HeteroData` object from a dictionary.

RETURN TYPE:

`Self`

property **stores**: *List[BaseStorage]*

Returns a list of all storages of the graph.

RETURN TYPE:

`List [ BaseStorage ]`

property **node_types**: *List[str]*

Returns a list of all node types of the graph.

RETURN TYPE:

`List [ str ]`

property **node_stores**: *List[NodeStorage]*

Returns a list of all node storages of the graph.

RETURN TYPE:

`List [ NodeStorage ]`

property **edge_types**: *List[Tuple[str, str, str]]*

Returns a list of all edge types of the graph.

RETURN TYPE:

`List [ Tuple [ str , str , str ]]`

property **edge_stores**: *List[EdgeStorage]*

Returns a list of all edge storages of the graph.

RETURN TYPE:

`List [ EdgeStorage ]`

node_items ( ) → List[Tuple[str, NodeStorage]] [source]

Returns a list of node type and node storage pairs.

RETURN TYPE:

`List [ Tuple [ str , NodeStorage ]]`

edge_items ( ) → List[Tuple[Tuple[str, str, str], EdgeStorage]] [source]

Returns a list of edge type and edge storage pairs.

RETURN TYPE:

`List [ Tuple [ Tuple [ str , str , str ], EdgeStorage ]]`

to_dict ( ) → Dict[str, Any] [source]

Returns a dictionary of stored key/value pairs.

RETURN TYPE:

`Dict [ str , Any ]`

to_namedtuple ( ) → NamedTuple [source]

Returns a `NamedTuple` of stored key/value pairs.

RETURN TYPE:

`NamedTuple`

set_value_dict (**key**: str, **value_dict**: Dict[str, Any]) → Self [source]

Sets the values in the dictionary `value_dict` to the attribute with name `key` to all node/edge types present in the dictionary.

```
data = HeteroData()

data.set_value_dict('x', {
    'paper': torch.randn(4, 16),
    'author': torch.randn(8, 32),
})

print(data['paper'].x)
```

RETURN TYPE:

`Self`

update (**data**: Self) → Self [source]

Updates the data object with the elements from another data object. Added elements will override existing ones (in case of duplicates).

RETURN TYPE:

`Self`

__cat_dim__ (**key**: str, **value**: Any, **store**: Optional[Union[NodeStorage, EdgeStorage]] = **None**, ***args**, ****kwargs**) → Any [source]

Returns the dimension for which the value `value` of the attribute `key` wil

concatenated when creating mini-batches using `torch_geometric.loader.DataLoader` .

Note

This method is for internal use only, and should only be overridden in case the mini-batch creation process is corrupted for a specific attribute.

RETURN TYPE:

`Any`

__inc__ (**key**: str, **value**: Any, **store**: Optional[Union[NodeStorage, EdgeStorage]] = **None**, ***args**, ****kwargs**) → Any [source]

Returns the incremental count to cumulatively increase the value `value` of the attribute `key` when creating mini-batches using `torch_geometric.loader.DataLoader`.

Note

This method is for internal use only, and should only be overridden in case the mini-batch creation process is corrupted for a specific attribute.

RETURN TYPE:

`Any`

property **num_nodes**: *Optional[int]*

Returns the number of nodes in the graph.

RETURN TYPE:

`Optional [ int ]`

property **num_node_features**: *Dict[str, int]*

Returns the number of features per node type in the graph.

RETURN TYPE:

`Dict [ str , int ]`

property **num_features**: *Dict[str, int]*

Returns the number of features per node type in the graph. Alias for `num_node_features`.

RETURN TYPE:

`Dict [ str , int ]`

property **num_edge_features**: *Dict[Tuple[str, str, str], int]*

Returns the number of features per edge type in the graph.

RETURN TYPE:

`Dict [ Tuple [ str , str , str ], int ]`

has_isolated_nodes ( ) → bool [source]

Returns `True` if the graph contains isolated nodes.

RETURN TYPE:

`bool`

is_undirected ( ) → bool [source]

Returns `True` if graph edges are undirected.

RETURN TYPE:

`bool`

validate (**raise_on_error**: bool = `True`) → bool [source]

Validates the correctness of the data.

RETURN TYPE:

`bool`

metadata ( ) → Tuple[List[str], List[Tuple[str, str, str]]] [source]

Returns the heterogeneous meta-data, *i.e.* its node and edge types.

```
data = HeteroData()
data['paper'].x = ...
data['author'].x = ...
data['author', 'writes', 'paper'].edge_index = ...

print(data.metadata())
>>> (['paper', 'author'], [('author', 'writes', 'paper')])
```

RETURN TYPE:

`Tuple [ List [ str ], List [ Tuple [ str , str , str ]]]`

collect (**key**: str, **allow_empty**: bool = `False`) → Dict[Union[str, Tuple[str, str, str]], Any] [source]

Collects the attribute `key` from all node and edge types.

```
data = HeteroData()
data['paper'].x = ...
data['author'].x = ...

print(data.collect('x'))
>>> { 'paper': ..., 'author': ...}
```

Note

This is equivalent to writing `data.x_dict` .

PARAMETERS:

- **key** (*str*) – The attribute to collect from all node and ege types.
- **allow_empty** (*bool*, *optional*) – If set to `True` , will not raise an error in case the attribute does not exit in any node or edge type. (default: `False`)

RETURN TYPE:

`Dict [ Union [ str , Tuple [ str , str , str ]], Any ]`

get_node_store (**key**: str) → NodeStorage [source]

Gets the `NodeStorage` object of a particular node type `key` . If the storage is not present yet, will create a new `torch_geometric.data.storage.NodeStorage` object for the given node type.

```
data = HeteroData()
node_storage = data.get_node_store('paper')
```

RETURN TYPE:

`NodeStorage`

get_edge_store (**src**: str, **rel**: str, **dst**: str) → EdgeStorage [source]

Gets the `EdgeStorage` object of a particular edge type given by the tuple `(src, rel, dst)`. If the storage is not present yet, will create a new `torch_geometric.data.storage.EdgeStorage` object for the given edge type.

```
data = HeteroData()
edge_storage = data.get_edge_store('author', 'writes', 'paper')
```

RETURN TYPE:

`EdgeStorage`

rename (**name**: str, **new_name**: str) → Self [source]

Renames the node type `name` to `new_name` in-place.

RETURN TYPE:

`Self`

subgraph (**subset_dict**: Dict[str, Tensor]) → Self [source]

Returns the induced subgraph containing the node types and corresponding nodes in `subset_dict`.

If a node type is not a key in `subset_dict` then all nodes of that type remain in the graph.

```
data = HeteroData()
data['paper'].x = ...
data['author'].x = ...
data['conference'].x = ...
data['paper', 'cites', 'paper'].edge_index = ...
data['author', 'paper'].edge_index = ...
data['paper', 'conference'].edge_index = ...
print(data)
>>> HeteroData(
    paper={ x=[10, 16] },
    author={ x=[5, 32] },
    conference={ x=[5, 8] },
    (paper, cites, paper)={ edge_index=[2, 50] },
    (author, to, paper)={ edge_index=[2, 30] },
    (paper, to, conference)={ edge_index=[2, 25] }
)

subset_dict = {
    'paper': torch.tensor([3, 4, 5, 6]),
    'author': torch.tensor([0, 2]),
}

print(data.subgraph(subset_dict))
>>> HeteroData(
    paper={ x=[4, 16] },
    author={ x=[2, 32] },
    conference={ x=[5, 8] },
    (paper, cites, paper)={ edge_index=[2, 24] },
    (author, to, paper)={ edge_index=[2, 5] },
    (paper, to, conference)={ edge_index=[2, 10] }
)
```

PARAMETERS:

subset_dict (*Dict[str, LongTensor or BoolTensor]*) – A dictionary holding the nodes to keep for each node type.

RETURN TYPE:

`Self`

edge_subgraph (**subset_dict**: Dict[Tuple[str, str, str], Tensor]) → Self [source]

Returns the induced subgraph given by the edge indices in `subset_dict` for certain edge types. Will currently preserve all the nodes in the graph, even if they are isolated after subgraph computation.

PARAMETERS:

subset_dict (*Dict[Tuple[str, str, str], LongTensor or BoolTensor]*) – A dictionary holding the edges to keep for each edge type.

RETURN TYPE:

`Self`

node_type_subgraph (**node_types**: List[str]) → Self [source]

Returns the subgraph induced by the given `node_types`, *i.e.* the returned `HeteroData` object only contains the node types which are included in `node_types`, and only contains the edge types where both end points are included in `node_types`.

RETURN TYPE:

`Self`

edge_type_subgraph (**edge_types**: List[Tuple[str, str, str]]) → Self [source]

Returns the subgraph induced by the given `edge_types`, *i.e.* the returned `HeteroData` object only contains the edge types which are included in `edge_types`, and only contains the node types of the end points which are included in `node_types`.

RETURN TYPE:

`Self`

to_homogeneous (**node_attrs**: Optional[List[str]] = None, **edge_attrs**: Optional[List[str]] = None, **add_node_type**: bool = True, **add_edge_type**: bool = True, **dummy_values**: bool = True) → Data [source]

Converts a `HeteroData` object to a homogeneous `Data` object. By default, all features with same feature dimensionality across different types will be merged into a single representation, unless otherwise specified via the `node_attrs` and `edge_attrs` arguments. Furthermore, attributes named `node_type` and `edge_type` will be added to the returned `Data` object, denoting node-level and edge-level vectors holding the node and edge type as integers, respectively.

PARAMETERS:

- **node_attrs** (*List[str]*, *optional*) – The node features to combine across all node types. These node features need to be of the same feature dimensionality. If set to `None`, will automatically determine which node features to combine. (default: `None`)
- **edge_attrs** (*List[str]*, *optional*) – The edge features to combine across all edge types. These edge features need to be of the same feature dimensionality. If set to `None`, will automatically determine which edge features to combine. (default: `None`)
- **add_node_type** (*bool*, *optional*) – If set to `False`, will not add the node-level vector `node_type` to the returned `Data` object. (default: `True`)
- **add_edge_type** (*bool*, *optional*) – If set to `False`, will not add the edge-level vector `edge_type` to the returned `Data` object. (default: `True`)
- **dummy_values** (*bool*, *optional*) – If set to `True`, will fill attributes of remaining types with dummy values. Dummy values are `NaN` for floating point attribute booleans, and `-1` for integers. (default: `True`)

RETURN TYPE:

`Data`

get_all_tensor_attrs ( ) → List[TensorAttr] [source]

Returns all registered tensor attributes.

RETURN TYPE:

`List [ TensorAttr ]`

get_all_edge_attrs ( ) → List[EdgeAttr] [source]

Returns all registered edge attributes.

RETURN TYPE:

`List [ EdgeAttr ]`

apply (**func**: Callable, ***args**: *str*)

Applies the function `func`, either to all attributes or only the ones given in `*args`.

apply_ (**func**: Callable, ***args**: *str*)

Applies the in-place function `func`, either to all attributes or only the ones given in `*args`.

clone (***args**: *str*)

Performs cloning of tensors, either for all attributes or only the ones given in `*args`.

coalesce ( ) → Self

Sorts and removes duplicated entries from edge indices `edge_index`.

RETURN TYPE:

`Self`

concat (**data**: Self) → Self

Concatenates `self` with another `data` object. All values needs to have matching shapes at non-concat dimensions.

RETURN TYPE:

`Self`

contiguous (***args**: *str*)

Ensures a contiguous memory layout, either for all attributes or only the c

`*args`.

coo (**edge_types**: Optional[List[Any]] = **None**, **store**: bool = **False**) → Tuple[Dict[Tuple[str, str, str], Tensor], Dict[Tuple[str, str, str], Tensor], Dict[Tuple[str, str, str], Optional[Tensor]]]

Returns the edge indices in the `GraphStore` in COO format.

PARAMETERS:

- **edge_types** (*List[Any]*, *optional*) – The edge types of edge indices to obtain. If set to `None`, will return the edge indices of all existing edge types. (default: `None`)
- **store** (*bool*, *optional*) – Whether to store converted edge indices in the `GraphStore`. (default: `False`)

RETURN TYPE:

`Tuple [ Dict [ Tuple [ str , str , str ], Tensor ], Dict [ Tuple [ str , str , str ], Tensor ], Dict [ Tuple [ str , str , str ], Optional [ Tensor ]]]`

cpu (***args**: *str*)

Copies attributes to CPU memory, either for all attributes or only the ones given in `*args`.

csc (**edge_types**: Optional[List[Any]] = **None**, **store**: bool = **False**) → Tuple[Dict[Tuple[str, str, str], Tensor], Dict[Tuple[str, str, str], Tensor], Dict[Tuple[str, str, str], Optional[Tensor]]]

Returns the edge indices in the `GraphStore` in CSC format.

PARAMETERS:

- **edge_types** (*List[Any]*, *optional*) – The edge types of edge indices to obtain. If set to `None`, will return the edge indices of all existing edge types. (default: `None`)
- **store** (*bool*, *optional*) – Whether to store converted edge indices in the `GraphStore`. (default: `False`)

RETURN TYPE:

`Tuple [ Dict [ Tuple [ str , str , str ], Tensor ], Dict [ Tuple [ str , str , str ], Tensor ], Dict [ Tuple [ str , str , str ], Optional [ Tensor ]]]`

csr (**edge_types**: Optional[List[Any]] = **None**, **store**: bool = **False**) → Tuple[Dict[Tuple[str, str, str], Tensor], Dict[Tuple[str, str, str], Tensor], Dict[Tuple[str, str, str], Optional[Tensor]]]

Returns the edge indices in the `GraphStore` in CSR format.

PARAMETERS:

- **edge_types** (*List[Any]*, *optional*) – The edge types of edge indices to obtain. If set to `None`, will return the edge indices of all existing edge types. (default: `None`)
- **store** (*bool*, *optional*) – Whether to store converted edge indices in the `GraphStore`. (default: `False`)

RETURN TYPE:

`Tuple [ Dict [ Tuple [ str , str , str ], Tensor ], Dict [ Tuple [ str , str , str ] Dict [ Tuple [ str , str , str ], Optional [ Tensor ]]]`

cuda (**device**: Optional[Union[int, str]] = None, ***args**: *str*, **non_blocking**: bool = False)

Copies attributes to CUDA memory, either for all attributes or only the ones given in `*args` .

detach (***args**: *str*)

Detaches attributes from the computation graph by creating a new tensor, either for all attributes or only the ones given in `*args` .

detach_ (***args**: *str*)

Detaches attributes from the computation graph, either for all attributes or only the ones given in `*args` .

edge_attrs ( ) → List[str]

Returns all edge-level tensor attribute names.

RETURN TYPE:

`List [ str ]`

generate_ids ( )

Generates and sets `n_id` and `e_id` attributes to assign each node and edge to a continuously ascending and unique ID.

get_edge_index (***args**, ****kwargs**) → Tuple[Tensor, Tensor]

Synchronously obtains an `edge_index` tuple from the `GraphStore` .

PARAMETERS:

- ***args** – Arguments passed to `EdgeAttr` .
- ****kwargs** – Keyword arguments passed to `EdgeAttr` .

RAISES:

KeyError – If the `edge_index` corresponding to the input `EdgeAttr` was not found.

RETURN TYPE:

`Tuple [ Tensor , Tensor ]`

get_tensor (***args**, **convert_type**: bool = False, ****kwargs**) → Union[Tensor, ndarray]

Synchronously obtains a `tensor` from the `FeatureStore` .

PARAMETERS:

- ***args** – Arguments passed to `TensorAttr` .

- **convert_type** (*bool*, *optional*) – Whether to convert the type of the output tensor to the type of the attribute index. (default: `False`)
- ****kwargs** – Keyword arguments passed to `TensorAttr`.

RAISES:

ValueError – If the input `TensorAttr` is not fully specified.

RETURN TYPE:

`Union[Tensor, ndarray]`

get_tensor_size(*args, **kwargs) → Optional[Tuple[int, ...]]

Obtains the size of a tensor given its `TensorAttr`, or `None` if the tensor does not exist.

RETURN TYPE:

`Optional[Tuple[int, ...]]`

has_self_loops( ) → bool

Returns `True` if the graph contains self-loops.

RETURN TYPE:

`bool`

is_coalesced( ) → bool

Returns `True` if edge indices `edge_index` are sorted and do not contain duplicate entries.

RETURN TYPE:

`bool`

property **is_cuda**: *bool*

Returns `True` if any `torch.Tensor` attribute is stored on the GPU, `False` otherwise.

RETURN TYPE:

`bool`

is_directed( ) → bool

Returns `True` if graph edges are directed.

RETURN TYPE:

`bool`

is_sorted(**sort_by_row**: bool = `True`) → bool

Returns `True` if edge indices `edge_index` are sorted.

PARAMETERS:

sort_by_row (*bool*, *optional*) – If set to `False`, will require column-wise order/by destination node order of `edge_index`. (default: `True`)

RETURN TYPE:

`bool`

is_sorted_by_time( ) → bool

Returns `True` if `time` is sorted.

RETURN TYPE:

`bool`

keys( ) → List[str]

Returns a list of all graph attribute names.

RETURN TYPE:

`List [ str ]`

multi_get_tensor(**attrs**: List[TensorAttr], **convert_type**: bool = `False`) → List[Union[Tensor, ndarray]]

Synchronously obtains a list of tensors from the `FeatureStore` for each tensor associated with the attributes in `attrs`.

Note

The default implementation simply iterates over all calls to `get_tensor()`. Implementor classes that can provide additional, more performant functionality are recommended to to override this method.

PARAMETERS:

- **attrs** (*List[TensorAttr]*) – A list of input `TensorAttr` objects that identify the tensors to obtain.
- **convert_type** (*bool*, *optional*) – Whether to convert the type of the output tensor to the type of the attribute index. (default: `False`)

RAISES:

ValueError – If any input `TensorAttr` is not fully specified.

RETURN TYPE:

`List [ Union [ Tensor , ndarray ]]`

node_attrs( ) → List[str]

Returns all node-level tensor attribute names.

RETURN TYPE:

`List [ str ]`

property **num_edges**: *int*

Returns the number of edges in the graph. For undirected graphs, this will return the number of bi-directional edges, which is double the amount of unique edges.

RETURN TYPE:

`int`

pin_memory (***args**: *str*)

Copies attributes to pinned memory, either for all attributes or only the ones given in `*args` .

put_edge_index (**edge_index**: Tuple[Tensor, Tensor], ***args**, ****kwargs**) → bool

Synchronously adds an `edge_index` tuple to the `GraphStore` . Returns whether insertion was successful.

PARAMETERS:

- **edge_index** (*Tuple[torch.Tensor, torch.Tensor]*) – The `edge_index` tuple in a format specified in `EdgeAttr` .
- ***args** – Arguments passed to `EdgeAttr` .
- ****kwargs** – Keyword arguments passed to `EdgeAttr` .

RETURN TYPE:

`bool`

put_tensor (**tensor**: Union[Tensor, ndarray], ***args**, ****kwargs**) → bool

Synchronously adds a `tensor` to the `FeatureStore` . Returns whether insertion was successful.

PARAMETERS:

- **tensor** (*torch.Tensor or np.ndarray*) – The feature tensor to be added.
- ***args** – Arguments passed to `TensorAttr` .
- ****kwargs** – Keyword arguments passed to `TensorAttr` .

RAISES:

ValueError – If the input `TensorAttr` is not fully specified.

RETURN TYPE:

`bool`

record_stream (**stream**: Stream, ***args**: *str*)

Ensures that the tensor memory is not reused for another tensor until all current work queued on `stream` has been completed, either for all attributes or only the ones given in `*args`.

remove_edge_index (*args, **kwargs) → bool

Synchronously deletes an `edge_index` tuple from the `GraphStore`. Returns whether deletion was successful.

PARAMETERS:

- ***args** – Arguments passed to `EdgeAttr`.
- ****kwargs** – Keyword arguments passed to `EdgeAttr`.

RETURN TYPE:

`bool`

remove_tensor (*args, **kwargs) → bool

Removes a tensor from the `FeatureStore`. Returns whether deletion was successful.

PARAMETERS:

- ***args** – Arguments passed to `TensorAttr`.
- ****kwargs** – Keyword arguments passed to `TensorAttr`.

RAISES:

ValueError – If the input `TensorAttr` is not fully specified.

RETURN TYPE:

`bool`

requires_grad_ (*args: *str*, **requires_grad**: bool = `True`)

Tracks gradient computation, either for all attributes or only the ones given in `*args`.

share_memory_ (*args: *str*)

Moves attributes to shared memory, either for all attributes or only the ones given in `*args`.

size (**dim**: Optional[int] = `None`) → Optional[Union[Tuple[Optional[int], Optional[int]], int]]

Returns the size of the adjacency matrix induced by the graph.

RETURN TYPE:

`Union [ Tuple [ Optional [ int ], Optional [ int ]], int , None ]`

snapshot (**start_time**: Union[float, int], **end_time**: Union[float, int], **attr**: str = 't…

Returns a snapshot of `data` to only hold events that occurred in period `[start_time, end_time]`.

RETURN TYPE:

`Self`

sort (**sort_by_row**: bool = `True`) → Self

Sorts edge indices `edge_index` and their corresponding edge features.

PARAMETERS:

sort_by_row (*bool*, *optional*) – If set to `False`, will sort `edge_index` in column-wise order/by destination node. (default: `True`)

RETURN TYPE:

`Self`

sort_by_time ( ) → Self

Sorts data associated with `time` according to `time`.

RETURN TYPE:

`Self`

to (**device**: Union[int, str], ***args**: *str*, **non_blocking**: bool = `False`)

Performs tensor device conversion, either for all attributes or only the ones given in `*args`.

up_to (**end_time**: Union[float, int]) → Self

Returns a snapshot of `data` to only hold events that occurred up to `end_time` (inclusive of `edge_time`).

RETURN TYPE:

`Self`

update_tensor (**tensor**: Union[Tensor, ndarray], ***args**, ****kwargs**) → bool

Updates a `tensor` in the `FeatureStore` with a new value. Returns whether the update was succesful.

Note

Implementor classes can choose to define more efficient update metho(performs a removal and insertion.

PARAMETERS:

- **tensor** (*torch.Tensor or np.ndarray*) – The feature tensor to be updated.
- ***args** – Arguments passed to `TensorAttr`.
- ****kwargs** – Keyword arguments passed to `TensorAttr`.

RETURN TYPE:

`bool`

view (***args**, ****kwargs**) → AttrView

Returns a view of the `FeatureStore` given a not yet fully-specified `TensorAttr`.

RETURN TYPE:

`AttrView`