Session API

Session and sessionmaker()

class sqlalchemy.orm.session.**sessionmaker**(*bind=None*, *class_=<class 'sqlalchemy.orm.session.Session'>*, *autoflush=True*, *autocommit=False*, *expire_on_commit=True*, *info=None*, ***kw*)

Bases: sqlalchemy.orm.session._SessionClassMethods

A configurable Session factory.

The sessionmaker factory generates new Session objects when called, creating them given the configurational arguments established here.

e.g.:

```
# global scope
Session = sessionmaker(autoflush=False)

# later, in a local scope, create and use a session:
sess = Session()
```

Any keyword arguments sent to the constructor itself will override the “configured” keywords:

```
Session = sessionmaker()

# bind an individual session to a connection
sess = Session(bind=connection)
```

The class also includes a method configure(), which can be used to specify additional keyword arguments to the factory, which will take effect for subsequent Session objects generated. This is usually used to associate one or more Engine objects with an existing sessionmaker factory before it is first used:

```
# application starts
Session = sessionmaker()

# ... later
engine = create_engine('sqlite:///foo.db')
Session.configure(bind=engine)

sess = Session()
```

__call__(***local_kw*)

Produce a new Session object using the configuration established in this sessionmaker.

In Python, the `__call__` method is invoked on an object when it is "called" in the same way as a function:

```
Session = sessionmaker()
session = Session()  # invokes sessionmaker.__call__()
```

`__init__`(*bind=None*, *class_=<class 'sqlalchemy.orm.session.Session'>*, *autoflush=True*, *autocommit=False*, *expire_on_commit=True*, *info=None*, ***kw*)

Construct a new `sessionmaker`.

All arguments here except for `class_` correspond to arguments accepted by `Session` directly. See the `Session.__init__()` docstring for more details on parameters.

Parameters:

- **bind** – a `Engine` or other `Connectable` with which newly created `Session` objects will be associated.
- **class_** – class to use in order to create new `Session` objects. Defaults to `Session`.
- **autoflush** – The autoflush setting to use with newly created `Session` objects.
- **autocommit** – The autocommit setting to use with newly created `Session` objects.
- **expire_on_commit=True** – the expire_on_commit setting to use with newly created `Session` objects.
- **info** –

  optional dictionary of information that will be available via `Session.info`. Note this dictionary is *updated*, not replaced, when the `info` parameter is specified to the specific `Session` construction operation.

  New in version 0.9.0.

- ****kw** – all other keyword arguments are passed to the constructor of newly created `Session` objects.

`close_all`()

inherited from the `close_all()` method of `_SessionClassMethods`

Close *all* sessions in memory.

`configure`(***new_kw*)

(Re)configure the arguments for this sessionmaker.

e.g.:

```
Session = sessionmaker()

Session.configure(bind=create_engine('sqlite://'))
```

`identity_key`(**args*, ***kwargs*)

inherited from the `identity_key()` method of `_SessionClassMethods`

Return an identity key.

This is an alias of `util.identity_key()`.

object_session(*instance*)

inherited from the `object_session()` *method of* `_SessionClassMethods`

Return the `Session` to which an object belongs.

This is an alias of `object_session()`.

class `sqlalchemy.orm.session.` **Session**(*bind=None*, *autoflush=True*, *expire_on_commit=True*, *_enable_transaction_accounting=True*, *autocommit=False*, *twophase=False*, *weak_identity_map=True*, *binds=None*, *extension=None*, *enable_baked_queries=True*, *info=None*, *query_cls=<class 'sqlalchemy.orm.query.Query'>*)

Bases: `sqlalchemy.orm.session._SessionClassMethods`

Manages persistence operations for ORM-mapped objects.

The Session's usage paradigm is described at Using the Session.

__init__(*bind=None*, *autoflush=True*, *expire_on_commit=True*, *_enable_transaction_accounting=True*, *autocommit=False*, *twophase=False*, *weak_identity_map=True*, *binds=None*, *extension=None*, *enable_baked_queries=True*, *info=None*, *query_cls=<class 'sqlalchemy.orm.query.Query'>*)

Construct a new Session.

See also the `sessionmaker` function which is used to generate a `Session`-producing callable with a given set of arguments.

Parameters:

- **autocommit** –

  Warning

  The autocommit flag is **not for general use**, and if it is used, queries should only be invoked within the span of a `Session.begin()` / `Session.commit()` pair. Executing queries outside of a demarcated transaction is a legacy mode of usage, and can in some cases lead to concurrent connection checkouts.

  Defaults to `False`. When `True`, the `Session` does not keep a persistent transaction running, and will acquire connections from the engine on an as-needed basis, returning them immediately after their use. Flushes will begin and commit (or possibly rollback) their own transaction if no transaction is present. When using this mode, the `Session.begin()` method is used to explicitly start transactions.

  See also

  Autocommit Mode

- **autoflush** – When `True`, all query operations will issue a `flush()` call to this `Session` before proceeding. This is a convenience feature so that `flush()` need not be called repeatedly in order for database queries to retrieve results. It's typical

that `autoflush` is used in conjunction with `autocommit=False`. In this scenario, explicit calls to `flush()` are rarely needed; you usually only need to call `commit()` (which flushes) to finalize changes.

- **bind** – An optional `Engine` or `Connection` to which this `Session` should be bound. When specified, all SQL operations performed by this session will execute via this connectable.
- **binds** –

  An optional dictionary which contains more granular
  "bind" information than the `bind` parameter provides. This dictionary can map individual :class`.Table` instances as well as `Mapper` instances to individual `Engine` or `Connection` objects. Operations which proceed relative to a particular `Mapper` will consult this dictionary for the direct `Mapper` instance as well as the mapper's `mapped_table` attribute in order to locate a connectable to use. The full resolution is described in the `Session.get_bind()`. Usage looks like:

  ```
  Session = sessionmaker(binds={
      SomeMappedClass: create_engine('postgresql://engine1'),
      somemapper: create_engine('postgresql://engine2'),
      some_table: create_engine('postgresql://engine3'),
      })
  ```

  Also see the `Session.bind_mapper()` and `Session.bind_table()` methods.

- **class_** – Specify an alternate class other than `sqlalchemy.orm.session.Session` which should be used by the returned class. This is the only argument that is local to the `sessionmaker` function, and is not sent directly to the constructor for `Session`.
- **enable_baked_queries** –

  defaults to `True`. A flag consumed by the `sqlalchemy.ext.baked` extension to determine if "baked queries" should be cached, as is the normal operation of this extension. When set to `False`, all caching is disabled, including baked queries defined by the calling application as well as those used internally. Setting this flag to `False` can significantly reduce memory use, however will also degrade performance for those areas that make use of baked queries (such as relationship loaders). Additionally, baked query logic in the calling application or potentially within the ORM that may be malfunctioning due to cache key collisions or similar can be flagged by observing if this flag resolves the issue.

  New in version 1.2.

- **_enable_transaction_accounting** – Defaults to `True`. A legacy-only flag which when `False` disables *all* 0.5-style object accounting on transaction boundaries, including auto-expiry of instances on rollback and commit, maintenance of the "new" and "deleted" lists upon rollback, and autoflush of pending changes upon `begin()`, all of which are interdependent.
- **expire_on_commit** – Defaults to `True`. When `True`, all instances will be fully expired after each `commit()`, so that all attribute/object access subsequent to a completed transaction will load from the most recent database state.
- **extension** – An optional `SessionExtension` instance, or a list of such instances, which will receive pre- and post- commit and flush events, as well as a post-rollback event. **Deprecated.** Please see `SessionEvents`.
- **info** –

optional dictionary of arbitrary data to be associated with this `Session`. Is available via the `Session.info` attribute. Note the dictionary is copied at construction time so that modifications to the per- `Session` dictionary will be local to that `Session`.

New in version 0.9.0.

- **query_cls** – Class which should be used to create new Query objects, as returned by the `query()` method. Defaults to `Query`.
- **twophase** – When `True`, all transactions will be started as a "two phase" transaction, i.e. using the "two phase" semantics of the database in use along with an XID. During a `commit()`, after `flush()` has been issued for all attached databases, the `prepare()` method on each database's `TwoPhaseTransaction` will be called. This allows each database to roll back the entire transaction, before each transaction is committed.
- **weak_identity_map** – Defaults to `True` - when set to `False`, objects placed in the `Session` will be strongly referenced until explicitly removed or the `Session` is closed. **Deprecated** - The strong reference identity map is legacy. See the recipe at Session Referencing Behavior for an event-based approach to maintaining strong identity references.

add(*instance*, *_warn=True*)

Place an object in the `Session`.

Its state will be persisted to the database on the next flush operation.

Repeated calls to `add()` will be ignored. The opposite of `add()` is `expunge()`.

add_all(*instances*)

Add the given collection of instances to this `Session`.

begin(*subtransactions=False*, *nested=False*)

Begin a transaction on this `Session`.

The `Session.begin()` method is only meaningful if this session is in **autocommit mode** prior to it being called; see Autocommit Mode for background on this setting.

The method will raise an error if this `Session` is already inside of a transaction, unless `Session.begin.subtransactions` or `Session.begin.nested` are specified.

Parameters:

- **subtransactions** – if True, indicates that this `begin()` can create a subtransaction if a transaction is already in progress. For documentation on subtransactions, please see Using Subtransactions with Autocommit.
- **nested** – if True, begins a SAVEPOINT transaction and is equivalent to calling `begin_nested()`. For documentation on SAVEPOINT transactions, please see Using SAVEPOINT.

Returns:

the `SessionTransaction` object. Note that `SessionTransaction` acts as a Python context manager, allowing `Session.begin()` to be used in a "with" block.

See Autocommit Mode for an example.

See also

Autocommit Mode

`Session.begin_nested()`

`begin_nested`()

Begin a “nested” transaction on this Session, e.g. SAVEPOINT.

The target database(s) and associated drivers must support SQL SAVEPOINT for this method to function correctly.

For documentation on SAVEPOINT transactions, please see Using SAVEPOINT.

Returns:
the `SessionTransaction` object. Note that `SessionTransaction` acts as a context manager, allowing `Session.begin_nested()` to be used in a “with” block. See Using SAVEPOINT for a usage example.

See also

Using SAVEPOINT

Serializable isolation / Savepoints / Transactional DDL - special workarounds required with the SQLite driver in order for SAVEPOINT to work correctly.

`bind_mapper`(*mapper*, *bind*)

Associate a `Mapper` with a “bind”, e.g. a `Engine` or `Connection`.

The given mapper is added to a lookup used by the `Session.get_bind()` method.

`bind_table`(*table*, *bind*)

Associate a `Table` with a “bind”, e.g. a `Engine` or `Connection`.

The given mapper is added to a lookup used by the `Session.get_bind()` method.

`bulk_insert_mappings`(*mapper*, *mappings*, *return_defaults=False*, *render_nulls=False*)

Perform a bulk insert of the given list of mapping dictionaries.

The bulk insert feature allows plain Python dictionaries to be used as the source of simple INSERT operations which can be more easily grouped together into higher performing “executemany” operations. Using dictionaries, there is no “history” or session state management features in use, reducing latency when inserting large numbers of simple rows.

The values within the dictionaries as given are typically passed without modification into Core `Insert()` constructs, after organizing the values within them across the tables to which the given mapper is mapped.

New in version 1.0.0.

Warning

The bulk insert feature allows for a lower-latency INSERT of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw INSERT of records.

Please read the list of caveats at Bulk Operations before using this method, and fully test and confirm the functionality of all code developed using these systems.

Parameters:

- **mapper** – a mapped class, or the actual `Mapper` object, representing the single kind of object represented within the mapping list.
- **mappings** – a list of dictionaries, each one containing the state of the mapped row to be inserted, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary must contain all keys to be populated into all tables.
- **return_defaults** – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted **one at a time**, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however, `Session.bulk_insert_mappings.return_defaults` **greatly reduces the performance gains** of the method overall. If the rows to be inserted only refer to a single table, then there is no reason this flag should be set as the returned default information is not used.
- **render_nulls** –

  When True, a value of `None` will result in a NULL value being included in the INSERT statement, rather than the column being omitted from the INSERT. This allows all the rows being INSERTed to have the identical set of columns which allows the full set of rows to be batched to the DBAPI. Normally, each column-set that contains a different combination of NULL values than the previous row must omit a different series of columns from the rendered INSERT statement, which means it must be emitted as a separate statement. By passing this flag, the full set of rows are guaranteed to be batchable into one batch; the cost however is that server-side defaults which are invoked by an omitted column will be skipped, so care must be taken to ensure that these are not necessary.

  Warning

  When this flag is set, **server side default SQL values will not be invoked** for those columns that are inserted as NULL; the NULL value will be sent explicitly. Care must be taken to ensure that no server-side default functions need to be invoked for the operation as a whole.

  New in version 1.1.

See also

Bulk Operations

```
Session.bulk_save_objects()

Session.bulk_update_mappings()
```

bulk_save_objects(*objects*, *return_defaults=False*, *update_changed_only=True*)

Perform a bulk save of the given list of objects.

The bulk save feature allows mapped objects to be used as the source of simple INSERT and UPDATE operations which can be more easily grouped together into higher performing “executemany” operations; the extraction of data from the objects is also performed using a lower-latency process that ignores whether or not attributes have actually been modified in the case of UPDATEs, and also ignores SQL expressions.

The objects as given are not added to the session and no additional state is established on them, unless the `return_defaults` flag is also set, in which case primary key attributes and server-side default values will be populated.

New in version 1.0.0.

Warning

The bulk save feature allows for a lower-latency INSERT/UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw INSERT/UPDATES of records.

Please read the list of caveats at Bulk Operations before using this method, and fully test and confirm the functionality of all code developed using these systems.

Parameters:

- **objects** –

  a list of mapped object instances. The mapped objects are persisted as is, and are **not** associated with the `Session` afterwards.

  For each object, whether the object is sent as an INSERT or an UPDATE is dependent on the same rules used by the `Session` in traditional operation; if the object has the `InstanceState.key` attribute set, then the object is assumed to be “detached” and will result in an UPDATE. Otherwise, an INSERT is used.

  In the case of an UPDATE, statements are grouped based on which attributes have changed, and are thus to be the subject of each SET clause. If `update_changed_only` is False, then all attributes present within each object are applied to the UPDATE statement, which may help in allowing the statements to be grouped together into a larger executemany(), and will also reduce the overhead of checking history on attributes.

- **return_defaults** – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted **one at a time**, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however,

`Session.bulk_save_objects.return_defaults` **greatly reduces the performance gains** of the method overall.

- **update_changed_only** – when True, UPDATE statements are rendered based on those attributes in each state that have logged changes. When False, all attributes present are rendered into the SET clause with the exception of primary key attributes.

See also

Bulk Operations

`Session.bulk_insert_mappings()`

`Session.bulk_update_mappings()`

`bulk_update_mappings`(*mapper*, *mappings*)

Perform a bulk update of the given list of mapping dictionaries.

The bulk update feature allows plain Python dictionaries to be used as the source of simple UPDATE operations which can be more easily grouped together into higher performing "executemany" operations. Using dictionaries, there is no "history" or session state management features in use, reducing latency when updating large numbers of simple rows.

New in version 1.0.0.

Warning

The bulk update feature allows for a lower-latency UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw UPDATES of records.

Please read the list of caveats at Bulk Operations before using this method, and fully test and confirm the functionality of all code developed using these systems.

Parameters:

- **mapper** – a mapped class, or the actual `Mapper` object, representing the single kind of object represented within the mapping list.
- **mappings** – a list of dictionaries, each one containing the state of the mapped row to be updated, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary may contain keys corresponding to all tables. All those keys which are present and are not part of the primary key are applied to the SET clause of the UPDATE statement; the primary key values, which are required, are applied to the WHERE clause.

See also

Bulk Operations

`Session.bulk_insert_mappings()`

`Session.bulk_save_objects()`

close()

Close this Session.

This clears all items and ends any transaction in progress.

If this session were created with `autocommit=False`, a new transaction is immediately begun. Note that this new transaction does not use any connection resources until they are first needed.

close_all()

inherited from the `close_all()` *method of* `_SessionClassMethods`

Close *all* sessions in memory.

commit()

Flush pending changes and commit the current transaction.

If no transaction is in progress, this method raises an `InvalidRequestError`.

By default, the `Session` also expires all database loaded state on all ORM-managed attributes after transaction commit. This so that subsequent operations load the most recent data from the database. This behavior can be disabled using the `expire_on_commit=False` option to `sessionmaker` or the `Session` constructor.

If a subtransaction is in effect (which occurs when begin() is called multiple times), the subtransaction will be closed, and the next call to `commit()` will operate on the enclosing transaction.

When using the `Session` in its default mode of `autocommit=False`, a new transaction will be begun immediately after the commit, but note that the newly begun transaction does *not* use any connection resources until the first SQL is actually emitted.

See also

Committing

connection(*mapper=None*, *clause=None*, *bind=None*, *close_with_result=False*, *execution_options=None*, ***kw*)

Return a `Connection` object corresponding to this `Session` object's transactional state.

If this `Session` is configured with `autocommit=False`, either the `Connection` corresponding to the current transaction is returned, or if no transaction is in progress, a new one is begun and the `Connection` returned (note that no transactional state is established with the DBAPI until the first SQL statement is emitted).

Alternatively, if this `Session` is configured with `autocommit=True`, an ad-hoc `Connection` is returned using `Engine.contextual_connect()` on the underlying `Engine`.

Ambiguity in multi-bind or unbound `Session` objects can be resolved through any of the optional keyword arguments. This ultimately makes usage of the `get_bind()` method for resolution.

Parameters:

- **bind** – Optional `Engine` to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over `mapper`, `clause`.
- **mapper** – Optional `mapper()` mapped class, used to identify the appropriate bind. This argument takes precedence over `clause`.
- **clause** – A `ClauseElement` (i.e. `select()`, `text()`, etc.) which will be used to locate a bind, if a bind cannot otherwise be identified.
- **close_with_result** – Passed to `Engine.connect()`, indicating the `Connection` should be considered "single use", automatically closing when the first result set is closed. This flag only has an effect if this `Session` is configured with `autocommit=True` and does not already have a transaction in progress.
- **execution_options** –

  a dictionary of execution options that will be passed to `Connection.execution_options()`, **when the connection is first procured only**. If the connection is already present within the `Session`, a warning is emitted and the arguments are ignored.

  New in version 0.9.9.

  See also

  Setting Transaction Isolation Levels

- ****kw** – Additional keyword arguments are sent to `get_bind()`, allowing additional arguments to be passed to custom implementations of `get_bind()`.

`delete`(*instance*)

Mark an instance as deleted.

The database delete operation occurs upon `flush()`.

`deleted`

The set of all instances marked as 'deleted' within this `Session`

`dirty`

The set of all persistent instances considered dirty.

E.g.:

```
some_mapped_object in session.dirty
```

Instances are considered dirty when they were modified but not deleted.

Note that this 'dirty' calculation is 'optimistic'; most attribute-setting or collection modification operations will mark an instance as 'dirty' and place it in this set, even if there is no net change to the attribute's value. At flush time, the value of each

attribute is compared to its previously saved value, and if there's no net change, no SQL operation will occur (this is a more expensive operation so it's only done at flush time).

To check if an instance has actionable net changes to its attributes, use the `Session.is_modified()` method.

`enable_relationship_loading`(*obj*)

Associate an object with this `Session` for related object loading.

> **Warning**
>
> `enable_relationship_loading()` exists to serve special use cases and is not recommended for general use.

Accesses of attributes mapped with `relationship()` will attempt to load a value from the database using this `Session` as the source of connectivity. The values will be loaded based on foreign key values present on this object - it follows that this functionality generally only works for many-to-one-relationships.

The object will be attached to this session, but will **not** participate in any persistence operations; its state for almost all purposes will remain either "transient" or "detached", except for the case of relationship loading.

Also note that backrefs will often not work as expected. Altering a relationship-bound attribute on the target object may not fire off a backref event, if the effective value is what was already loaded from a foreign-key-holding value.

The `Session.enable_relationship_loading()` method is similar to the `load_on_pending` flag on `relationship()`. Unlike that flag, `Session.enable_relationship_loading()` allows an object to remain transient while still being able to load related items.

To make a transient object associated with a `Session` via `Session.enable_relationship_loading()` pending, add it to the `Session` using `Session.add()` normally.

`Session.enable_relationship_loading()` does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before flush() proceeds. This method is not intended for general use.

> *New in version 0.8.*

> **See also**
>
> `load_on_pending` at `relationship()` - this flag allows per-relationship loading of many-to-ones on items that are pending.

`execute`(*clause*, *params=None*, *mapper=None*, *bind=None*, ***kw*)

Execute a SQL expression construct or string statement within the current transaction.

Returns a `ResultProxy` representing results of the statement execution, in the same manner as that of an `Engine` or `Connection`.

E.g.:

```
result = session.execute(
            user_table.select().where(user_table.c.id == 5)
        )
```

`execute()` accepts any executable clause construct, such as `select()`, `insert()`, `update()`, `delete()`, and `text()`. Plain SQL strings can be passed as well, which in the case of `Session.execute()` only will be interpreted the same as if it were passed via a `text()` construct. That is, the following usage:

```
result = session.execute(
            "SELECT * FROM user WHERE id=:param",
            {"param":5}
        )
```

is equivalent to:

```
from sqlalchemy import text
result = session.execute(
            text("SELECT * FROM user WHERE id=:param"),
            {"param":5}
        )
```

The second positional argument to `Session.execute()` is an optional parameter set. Similar to that of `Connection.execute()`, whether this is passed as a single dictionary, or a list of dictionaries, determines whether the DBAPI cursor's `execute()` or `executemany()` is used to execute the statement. An INSERT construct may be invoked for a single row:

```
result = session.execute(
    users.insert(), {"id": 7, "name": "somename"})
```

or for multiple rows:

```
result = session.execute(users.insert(), [
                        {"id": 7, "name": "somename7"},
                        {"id": 8, "name": "somename8"},
                        {"id": 9, "name": "somename9"}
                    ])
```

The statement is executed within the current transactional context of this `Session`. The `Connection` which is used to execute the statement can also be acquired directly by calling the `Session.connection()` method. Both methods use a rule-based resolution scheme in order to determine the `Connection`, which in the average case is derived directly from the "bind" of the `Session` itself, and in other cases can be based on the `mapper()` and `Table` objects passed to the method; see the documentation for `Session.get_bind()` for a full description of this scheme.

The `Session.execute()` method does *not* invoke autoflush.

The `ResultProxy` returned by the `Session.execute()` method is returned with the "close_with_result" flag set to true; the significance of this flag is that if this `Session` is autocommitting and does not have a transaction-dedicated

`Connection` available, a temporary `Connection` is established for the statement execution, which is closed (meaning, returned to the connection pool) when the `ResultProxy` has consumed all available data. This applies *only* when the `Session` is configured with autocommit=True and no transaction has been started.

Parameters:

- **clause** – An executable statement (i.e. an `Executable` expression such as `expression.select()`) or string SQL statement to be executed.
- **params** – Optional dictionary, or list of dictionaries, containing bound parameter values. If a single dictionary, single-row execution occurs; if a list of dictionaries, an "executemany" will be invoked. The keys in each dictionary must correspond to parameter names present in the statement.
- **mapper** – Optional `mapper()` or mapped class, used to identify the appropriate bind. This argument takes precedence over `clause` when locating a bind. See `Session.get_bind()` for more details.
- **bind** – Optional `Engine` to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over `mapper` and `clause` when locating a bind.
- ****kw** – Additional keyword arguments are sent to `Session.get_bind()` to allow extensibility of "bind" schemes.

See also

SQL Expression Language Tutorial - Tutorial on using Core SQL constructs.

Working with Engines and Connections - Further information on direct statement execution.

`Connection.execute()` - core level statement execution method, which is `Session.execute()` ultimately uses in order to execute the statement.

`expire`(*instance*, *attribute_names=None*)

Expire the attributes on an instance.

Marks the attributes of an instance as out of date. When an expired attribute is next accessed, a query will be issued to the `Session` object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

To expire all objects in the `Session` simultaneously, use `Session.expire_all()`.

The `Session` object's default behavior is to expire all state whenever the `Session.rollback()` or `Session.commit()` methods are called, so that new state can be loaded for the new transaction. For this reason, calling `Session.expire()` only makes sense for the specific case that a non-ORM SQL statement was emitted in the current transaction.

Parameters:

- **instance** – The instance to be refreshed.
- **attribute_names** – optional list of string attribute names indicating a subset of attributes to be expired.

See also

Refreshing / Expiring - introductory material

`Session.expire()`

`Session.refresh()`

`expire_all`()

Expires all persistent instances within this Session.

When any attributes on a persistent instance is next accessed, a query will be issued using the `Session` object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

To expire individual objects and individual attributes on those objects, use `Session.expire()`.

The `Session` object's default behavior is to expire all state whenever the `Session.rollback()` or `Session.commit()` methods are called, so that new state can be loaded for the new transaction. For this reason, calling `Session.expire_all()` should not be needed when autocommit is `False`, assuming the transaction is isolated.

See also

Refreshing / Expiring - introductory material

`Session.expire()`

`Session.refresh()`

`expunge`(*instance*)

Remove the *instance* from this `Session`.

This will free all internal references to the instance. Cascading will be applied according to the *expunge* cascade rule.

`expunge_all`()

Remove all object instances from this `Session`.

This is equivalent to calling `expunge(obj)` on all objects in this `Session`.

`flush`(*objects=None*)

Flush all the object changes to the database.

Writes out all pending object creations, deletions and modifications to the database as INSERTs, DELETEs, UPDATEs, etc. Operations are automatically ordered by the Session's unit of work dependency solver.

Database operations will be issued in the current transactional context and do not affect the state of the transaction, unless an error occurs, in which case the entire

transaction is rolled back. You may flush() as often as you like within a transaction to move changes from Python to the database's transaction buffer.

For `autocommit` Sessions with no active manual transaction, flush() will create a transaction on the fly that surrounds the entire set of operations into the flush.

Parameters:
objects –

Optional; restricts the flush operation to operate only on elements that are in the given collection.

This feature is for an extremely narrow set of use cases where particular objects may need to be operated upon before the full flush() occurs. It is not intended for general use.

`get_bind`(*mapper=None*, *clause=None*)

Return a "bind" to which this `Session` is bound.

The "bind" is usually an instance of `Engine`, except in the case where the `Session` has been explicitly bound directly to a `Connection`.

For a multiply-bound or unbound `Session`, the `mapper` or `clause` arguments are used to determine the appropriate bind to return.

Note that the "mapper" argument is usually present when `Session.get_bind()` is called via an ORM operation such as a `Session.query()`, each individual INSERT/UPDATE/DELETE operation within a `Session.flush()`, call, etc.

The order of resolution is:

1. if mapper given and session.binds is present, locate a bind based on mapper.
2. if clause given and session.binds is present, locate a bind based on `Table` objects found in the given clause present in session.binds.
3. if session.bind is present, return that.
4. if clause given, attempt to return a bind linked to the `MetaData` ultimately associated with the clause.
5. if mapper given, attempt to return a bind linked to the `MetaData` ultimately associated with the `Table` or other selectable to which the mapper is mapped.
6. No bind can be found, `UnboundExecutionError` is raised.

Parameters:

- **mapper** – Optional `mapper()` mapped class or instance of `Mapper`. The bind can be derived from a `Mapper` first by consulting the "binds" map associated with this `Session`, and secondly by consulting the `MetaData` associated with the `Table` to which the `Mapper` is mapped for a bind.
- **clause** – A `ClauseElement` (i.e. `select()`, `text()`, etc.). If the `mapper` argument is not present or could not produce a bind, the given expression construct will be searched for a bound element, typically a `Table` associated with bound `MetaData`.

`identity_key`(**args*, ***kwargs*)

inherited from the `identity_key()` *method of* `_SessionClassMethods`

Return an identity key.

This is an alias of `util.identity_key()`.

`identity_map` = *None*

A mapping of object identities to objects themselves.

Iterating through `Session.identity_map.values()` provides access to the full set of persistent objects (i.e., those that have row identity) currently in the session.

See also

`identity_key()` - helper function to produce the keys used in this dictionary.

`info`

A user-modifiable dictionary.

The initial value of this dictionary can be populated using the `info` argument to the `Session` constructor or `sessionmaker` constructor or factory methods. The dictionary here is always local to this `Session` and can be modified independently of all other `Session` objects.

New in version 0.9.0.

`invalidate`()

Close this Session, using connection invalidation.

This is a variant of `Session.close()` that will additionally ensure that the `Connection.invalidate()` method will be called on all `Connection` objects. This can be called when the database is known to be in a state where the connections are no longer safe to be used.

E.g.:

```
try:
    sess = Session()
    sess.add(User())
    sess.commit()
except gevent.Timeout:
    sess.invalidate()
    raise
except:
    sess.rollback()
    raise
```

This clears all items and ends any transaction in progress.

If this session were created with `autocommit=False`, a new transaction is immediately begun. Note that this new transaction does not use any connection resources until they are first needed.

New in version 0.9.9.

is_active

True if this Session is in "transaction mode" and is not in "partial rollback" state.

The Session in its default mode of autocommit=False is essentially always in "transaction mode", in that a SessionTransaction is associated with it as soon as it is instantiated. This SessionTransaction is immediately replaced with a new one as soon as it is ended, due to a rollback, commit, or close operation.

"Transaction mode" does *not* indicate whether or not actual database connection resources are in use; the SessionTransaction object coordinates among zero or more actual database transactions, and starts out with none, accumulating individual DBAPI connections as different data sources are used within its scope. The best way to track when a particular Session has actually begun to use DBAPI resources is to implement a listener using the SessionEvents.after_begin() method, which will deliver both the Session as well as the target Connection to a user-defined event listener.

The "partial rollback" state refers to when an "inner" transaction, typically used during a flush, encounters an error and emits a rollback of the DBAPI connection. At this point, the Session is in "partial rollback" and awaits for the user to call Session.rollback(), in order to close out the transaction stack. It is in this "partial rollback" period that the is_active flag returns False. After the call to Session.rollback(), the SessionTransaction is replaced with a new one and is_active returns True again.

When a Session is used in autocommit=True mode, the SessionTransaction is only instantiated within the scope of a flush call, or when Session.begin() is called. So is_active will always be False outside of a flush or Session.begin() block in this mode, and will be True within the Session.begin() block as long as it doesn't enter "partial rollback" state.

From all the above, it follows that the only purpose to this flag is for application frameworks that wish to detect is a "rollback" is necessary within a generic error handling routine, for Session objects that would otherwise be in "partial rollback" mode. In a typical integration case, this is also not necessary as it is standard practice to emit Session.rollback() unconditionally within the outermost exception catch.

To track the transactional state of a Session fully, use event listeners, primarily the SessionEvents.after_begin(), SessionEvents.after_commit(), SessionEvents.after_rollback() and related events.

is_modified(*instance*, *include_collections=True*, *passive=True*)

Return True if the given instance has locally modified attributes.

This method retrieves the history for each instrumented attribute on the instance and performs a comparison of the current value to its previously committed value, if any.

It is in effect a more expensive and accurate version of checking for the given instance in the Session.dirty collection; a full test for each attribute's net "dirty" status is performed.

E.g.:

```
return session.is_modified(someobject)
```

Changed in version 0.8: When using SQLAlchemy 0.7 and earlier, the `passive` flag should **always** be explicitly set to `True`, else SQL loads/autoflushes may proceed which can affect the modified state itself: `session.is_modified(someobject, passive=True)`. In 0.8 and above, the behavior is corrected and this flag is ignored.

A few caveats to this method apply:

- Instances present in the `Session.dirty` collection may report `False` when tested with this method. This is because the object may have received change events via attribute mutation, thus placing it in `Session.dirty`, but ultimately the state is the same as that loaded from the database, resulting in no net change here.
- Scalar attributes may not have recorded the previously set value when a new value was applied, if the attribute was not loaded, or was expired, at the time the new value was received - in these cases, the attribute is assumed to have a change, even if there is ultimately no net change against its database value. SQLAlchemy in most cases does not need the "old" value when a set event occurs, so it skips the expense of a SQL call if the old value isn't present, based on the assumption that an UPDATE of the scalar value is usually needed, and in those few cases where it isn't, is less expensive on average than issuing a defensive SELECT.

  The "old" value is fetched unconditionally upon set only if the attribute container has the `active_history` flag set to `True`. This flag is set typically for primary key attributes and scalar object references that are not a simple many-to-one. To set this flag for any arbitrary mapped column, use the `active_history` argument with `column_property()`.

Parameters:

- **instance** – mapped instance to be tested for pending changes.
- **include_collections** – Indicates if multivalued collections should be included in the operation. Setting this to `False` is a way to detect only local-column based properties (i.e. scalar columns or many-to-one foreign keys) that would result in an UPDATE for this instance upon flush.
- **passive** –

  Changed in version 0.8: Ignored for backwards compatibility. When using SQLAlchemy 0.7 and earlier, this flag should always be set to `True`.

`merge`(*instance*, *load=True*)

Copy the state of a given instance into a corresponding instance within this `Session`.

`Session.merge()` examines the primary key attributes of the source instance, and attempts to reconcile it with an instance of the same primary key in the session. If not found locally, it attempts to load the object from the database based on primary key, and if none can be located, creates a new instance. The state of each attribute on the source instance is then copied to the target instance. The resulting target instance is then returned by the method; the original source instance is left unmodified, and un-associated with the `Session` if not already.

This operation cascades to associated instances if the association is mapped with `cascade="merge"`.

See Merging for a detailed discussion of merging.

Changed in version 1.1: - `Session.merge()` will now reconcile pending objects with overlapping primary keys in the same way as persistent. See Session.merge resolves pending conflicts the same as persistent for discussion.

Parameters:

- **instance** – Instance to be merged.
- **load** –

  Boolean, when False, `merge()` switches into a "high performance" mode which causes it to forego emitting history events as well as all database access. This flag is used for cases such as transferring graphs of objects into a `Session` from a second level cache, or to transfer just-loaded objects into the `Session` owned by a worker thread or process without re-querying the database.

  The `load=False` use case adds the caveat that the given object has to be in a "clean" state, that is, has no pending changes to be flushed - even if the incoming object is detached from any `Session`. This is so that when the merge operation populates local attributes and cascades to related objects and collections, the values can be "stamped" onto the target object as is, without generating any history or attribute events, and without the need to reconcile the incoming data with any existing related objects or collections that might not be loaded. The resulting objects from `load=False` are always produced as "clean", so it is only appropriate that the given objects should be "clean" as well, else this suggests a mis-use of the method.

new

The set of all instances marked as 'new' within this `Session`.

no_autoflush

Return a context manager that disables autoflush.

e.g.:

```
with session.no_autoflush:

    some_object = SomeClass()
    session.add(some_object)
    # won't autoflush
    some_object.related_thing = session.query(SomeRelated).first()
```

Operations that proceed within the `with:` block will not be subject to flushes occurring upon query access. This is useful when initializing a series of objects which involve existing database queries, where the uncompleted object should not yet be flushed.

New in version 0.7.6.

object_session(*instance*)

inherited from the `object_session()` *method of* `_SessionClassMethods`

Return the `Session` to which an object belongs.

This is an alias of `object_session()`.

prepare()

Prepare the current transaction in progress for two phase commit.

If no transaction is in progress, this method raises an `InvalidRequestError`.

Only root transactions of two phase sessions can be prepared. If the current transaction is not such, an `InvalidRequestError` is raised.

prune()

Remove unreferenced instances cached in the identity map.

Deprecated since version 0.7: The non-weak-referencing identity map feature is no longer needed.

Note that this method is only meaningful if “weak_identity_map” is set to False. The default weak identity map is self-pruning.

Removes any object in this Session’s identity map that is not referenced in user code, modified, new or scheduled for deletion. Returns the number of objects pruned.

query(**entities*, ***kwargs*)

Return a new `Query` object corresponding to this `Session`.

refresh(*instance*, *attribute_names=None*, *with_for_update=None*, *lockmode=None*)

Expire and refresh the attributes on the given instance.

A query will be issued to the database and all attributes will be refreshed with their current database value.

Lazy-loaded relational attributes will remain lazily loaded, so that the instance-wide refresh operation will be followed immediately by the lazy load of that attribute.

Eagerly-loaded relational attributes will eagerly load within the single refresh operation.

Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction - usage of `refresh()` usually only makes sense if non-ORM SQL statement were emitted in the ongoing transaction, or if autocommit mode is turned on.

Parameters:

- **attribute_names** – optional. An iterable collection of string attribute names indicating a subset of attributes to be refreshed.

- **with_for_update** –

  optional boolean `True` indicating FOR UPDATE should be used, or may be a dictionary containing flags to indicate a more specific set of FOR UPDATE flags for the SELECT; flags should match the parameters of `Query.with_for_update()`. Supersedes the `Session.refresh.lockmode` parameter.

  New in version 1.2.

- **lockmode** – Passed to the `Query` as used by `with_lockmode()`. Superseded by `Session.refresh.with_for_update`.

See also

Refreshing / Expiring - introductory material

`Session.expire()`

`Session.expire_all()`

`rollback`()

Rollback the current transaction in progress.

If no transaction is in progress, this method is a pass-through.

This method rolls back the current transaction or nested transaction regardless of subtransactions being in effect. All subtransactions up to the first real transaction are closed. Subtransactions occur when `begin()` is called multiple times.

See also

Rolling Back

`scalar`(*clause*, *params=None*, *mapper=None*, *bind=None*, ***kw*)

Like `execute()` but return a scalar result.

`transaction` = *None*

The current active or inactive `SessionTransaction`.

class `sqlalchemy.orm.session.` **`SessionTransaction`**(*session*, *parent=None*, *nested=False*)

A `Session`-level transaction.

`SessionTransaction` is a mostly behind-the-scenes object not normally referenced directly by application code. It coordinates among multiple `Connection` objects, maintaining a database transaction for each one individually, committing or rolling them back all at once. It also provides optional two-phase commit behavior which can augment this coordination operation.

The `Session.transaction` attribute of `Session` refers to the current `SessionTransaction` object in use, if any. The `SessionTransaction.parent`

attribute refers to the parent `SessionTransaction` in the stack of `SessionTransaction` objects. If this attribute is `None`, then this is the top of the stack. If non-`None`, then this `SessionTransaction` refers either to a so-called "subtransaction" or a "nested" transaction. A "subtransaction" is a scoping concept that demarcates an inner portion of the outermost "real" transaction. A nested transaction, which is indicated when the `SessionTransaction.nested` attribute is also True, indicates that this `SessionTransaction` corresponds to a SAVEPOINT.

Life Cycle

A `SessionTransaction` is associated with a `Session` in its default mode of `autocommit=False` immediately, associated with no database connections. As the `Session` is called upon to emit SQL on behalf of various `Engine` or `Connection` objects, a corresponding `Connection` and associated `Transaction` is added to a collection within the `SessionTransaction` object, becoming one of the connection/transaction pairs maintained by the `SessionTransaction`. The start of a `SessionTransaction` can be tracked using the `SessionEvents.after_transaction_create()` event.

The lifespan of the `SessionTransaction` ends when the `Session.commit()`, `Session.rollback()` or `Session.close()` methods are called. At this point, the `SessionTransaction` removes its association with its parent `Session`. A `Session` that is in `autocommit=False` mode will create a new `SessionTransaction` to replace it immediately, whereas a `Session` that's in `autocommit=True` mode will remain without a `SessionTransaction` until the `Session.begin()` method is called. The end of a `SessionTransaction` can be tracked using the `SessionEvents.after_transaction_end()` event.

Nesting and Subtransactions

Another detail of `SessionTransaction` behavior is that it is capable of "nesting". This means that the `Session.begin()` method can be called while an existing `SessionTransaction` is already present, producing a new `SessionTransaction` that temporarily replaces the parent `SessionTransaction`. When a `SessionTransaction` is produced as nested, it assigns itself to the `Session.transaction` attribute, and it additionally will assign the previous `SessionTransaction` to its `Session.parent` attribute. The behavior is effectively a stack, where `Session.transaction` refers to the current head of the stack, and the `SessionTransaction.parent` attribute allows traversal up the stack until `SessionTransaction.parent` is `None`, indicating the top of the stack.

When the scope of `SessionTransaction` is ended via `Session.commit()` or `Session.rollback()`, it restores its parent `SessionTransaction` back onto the `Session.transaction` attribute.

The purpose of this stack is to allow nesting of `Session.rollback()` or `Session.commit()` calls in context with various flavors of `Session.begin()`. This nesting behavior applies to when `Session.begin_nested()` is used to emit a SAVEPOINT transaction, and is also used to produce a so-called "subtransaction" which allows a block of code to use a begin/rollback/commit sequence regardless of whether or not its enclosing code block has begun a transaction. The `flush()` method, whether called explicitly or via autoflush, is the primary consumer of the "subtransaction" feature, in that it wishes to guarantee that it works within in a transaction block regardless of whether or not the `Session` is in transactional mode when the method is called.

Note that the flush process that occurs within the "autoflush" feature as well as when the `Session.flush()` method is used **always** creates a `SessionTransaction` object. This object is normally a subtransaction, unless the `Session` is in autocommit mode and no transaction exists at all, in which case it's the outermost transaction. Any event-handling logic or other inspection logic needs to take into account whether a

`SessionTransaction` is the outermost transaction, a subtransaction, or a "nested" / SAVEPOINT transaction.

`Session.rollback()`

`Session.commit()`

`Session.begin()`

`Session.begin_nested()`

`Session.is_active`

`SessionEvents.after_transaction_create()`

`SessionEvents.after_transaction_end()`

`SessionEvents.after_commit()`

`SessionEvents.after_rollback()`

`SessionEvents.after_soft_rollback()`

`nested` = *False*

Indicates if this is a nested, or SAVEPOINT, transaction.

When `SessionTransaction.nested` is True, it is expected that `SessionTransaction.parent` will be True as well.

`parent`

The parent `SessionTransaction` of this `SessionTransaction`.

If this attribute is `None`, indicates this `SessionTransaction` is at the top of the stack, and corresponds to a real "COMMIT"/"ROLLBACK" block. If non-`None`, then this is either a "subtransaction" or a "nested" / SAVEPOINT transaction. If the `SessionTransaction.nested` attribute is `True`, then this is a SAVEPOINT, and if `False`, indicates this a subtransaction.

New in version 1.0.16: - use ._parent for previous versions

Session Utilities

`sqlalchemy.orm.session.`**`make_transient`**(*instance*)

Alter the state of the given instance so that it is transient.

Note

`make_transient()` is a special-case function for advanced use cases only.

The given mapped instance is assumed to be in the persistent or detached state. The function will remove its association with any `Session` as well as its `InstanceState.identity`. The effect is that the object will behave as though it were newly constructed, except retaining any attribute / collection values that were loaded at

the time of the call. The `InstanceState.deleted` flag is also reset if this object had been deleted as a result of using `Session.delete()`.

Warning

`make_transient()` does **not** "unexpire" or otherwise eagerly load ORM-mapped attributes that are not currently loaded at the time the function is called. This includes attributes which:

- were expired via `Session.expire()`
- were expired as the natural effect of committing a session transaction, e.g. `Session.commit()`
- are normally lazy loaded but are not currently loaded
- are "deferred" via Deferred Column Loading and are not yet loaded
- were not present in the query which loaded this object, such as that which is common in joined table inheritance and other scenarios.

After `make_transient()` is called, unloaded attributes such as those above will normally resolve to the value `None` when accessed, or an empty collection for a collection-oriented attribute. As the object is transient and un-associated with any database identity, it will no longer retrieve these values.

See also

`make_transient_to_detached()`

sqlalchemy.orm.session.**make_transient_to_detached**(*instance*)

Make the given transient instance detached.

Note

`make_transient_to_detached()` is a special-case function for advanced use cases only.

All attribute history on the given instance will be reset as though the instance were freshly loaded from a query. Missing attributes will be marked as expired. The primary key attributes of the object, which are required, will be made into the "key" of the instance.

The object can then be added to a session, or merged possibly with the load=False flag, at which point it will look as if it were loaded that way, without emitting SQL.

This is a special use case function that differs from a normal call to `Session.merge()` in that a given persistent state can be manufactured without any SQL calls.

New in version 0.9.5.

See also

`make_transient()`

sqlalchemy.orm.session.**object_session**(*instance*)

Return the `Session` to which the given instance belongs.

This is essentially the same as the `InstanceState.session` accessor. See that attribute for details.

`sqlalchemy.orm.util.`**`was_deleted`**(*object*)

Return True if the given object was deleted within a session flush.

This is regardless of whether or not the object is persistent or detached.

New in version 0.8.0.

See also

`InstanceState.was_deleted`

Attribute and State Management Utilities

These functions are provided by the SQLAlchemy attribute instrumentation API to provide a detailed interface for dealing with instances, attribute values, and history. Some of them are useful when constructing event listener functions, such as those described in ORM Events.

`sqlalchemy.orm.util.`**`object_state`**(*instance*)

Given an object, return the `InstanceState` associated with the object.

Raises `sqlalchemy.orm.exc.UnmappedInstanceError` if no mapping is configured.

Equivalent functionality is available via the `inspect()` function as:

```
inspect(instance)
```

Using the inspection system will raise `sqlalchemy.exc.NoInspectionAvailable` if the instance is not part of a mapping.

`sqlalchemy.orm.attributes.`**`del_attribute`**(*instance*, *key*)

Delete the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

`sqlalchemy.orm.attributes.`**`get_attribute`**(*instance*, *key*)

Get the value of an attribute, firing any callables required.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to make usage of attribute state as understood by SQLAlchemy.

`sqlalchemy.orm.attributes.`**`get_history`**(*obj*, *key*, *passive=symbol('PASSIVE_OFF')*)

Return a `History` record for the given object and attribute key.

Parameters:

- **obj** – an object whose class is instrumented by the attributes package.
- **key** – string attribute name.
- **passive** – indicates loading behavior for the attribute if the value is not already present. This is a bitflag attribute, which defaults to the symbol `PASSIVE_OFF` indicating all necessary SQL should be emitted.

sqlalchemy.orm.attributes.**init_collection**(*obj*, *key*)

Initialize a collection attribute and return the collection adapter.

This function is used to provide direct access to collection internals for a previously unloaded attribute. e.g.:

```
collection_adapter = init_collection(someobject, 'elements')
for elem in values:
    collection_adapter.append_without_event(elem)
```

For an easier way to do the above, see `set_committed_value()`.

obj is an instrumented object instance. An InstanceState is accepted directly for backwards compatibility but this usage is deprecated.

sqlalchemy.orm.attributes.**flag_modified**(*instance*, *key*)

Mark an attribute on an instance as 'modified'.

This sets the 'modified' flag on the instance and establishes an unconditional change event for the given attribute. The attribute must have a value present, else an `InvalidRequestError` is raised.

To mark an object "dirty" without referring to any specific attribute so that it is considered within a flush, use the `attributes.flag_dirty()` call.

See also

`attributes.flag_dirty()`

sqlalchemy.orm.attributes.**flag_dirty**(*instance*)

Mark an instance as 'dirty' without any specific attribute mentioned.

This is a special operation that will allow the object to travel through the flush process for interception by events such as `SessionEvents.before_flush()`. Note that no SQL will be emitted in the flush process for an object that has no changes, even if marked dirty via this method. However, a `SessionEvents.before_flush()` handler will be able to see the object in the `Session.dirty` collection and may establish changes on it, which will then be included in the SQL emitted.

New in version 1.2.

See also

```
attributes.flag_modified()
```

sqlalchemy.orm.attributes.**instance_state**()

Return the `InstanceState` for a given mapped object.

This function is the internal version of `object_state()`. The `object_state()` and/or the `inspect()` function is preferred here as they each emit an informative exception if the given object is not mapped.

sqlalchemy.orm.instrumentation.**is_instrumented**(*instance*, *key*)

Return True if the given attribute on the given instance is instrumented by the attributes package.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required.

sqlalchemy.orm.attributes.**set_attribute**(*instance*, *key*, *value*)

Set the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

sqlalchemy.orm.attributes.**set_committed_value**(*instance*, *key*, *value*)

Set the value of an attribute with no history events.

Cancels any previous history present. The value should be a scalar value for scalar-holding attributes, or an iterable for any collection-holding attribute.

This is the same underlying method used when a lazy loader fires off and loads additional data from the database. In particular, this method can be used by application code which has loaded additional attributes or collections through separate queries, which can then be attached to an instance as though it were part of its original loaded state.

class sqlalchemy.orm.attributes.**History**

Bases: `sqlalchemy.orm.attributes.History`

A 3-tuple of added, unchanged and deleted values, representing the changes which have occurred on an instrumented attribute.

The easiest way to get a `History` object for a particular attribute on an object is to use the `inspect()` function:

```
from sqlalchemy import inspect

hist = inspect(myobject).attrs.myattribute.history
```

Each tuple member is an iterable sequence:

- `added` - the collection of items added to the attribute (the first tuple element).
- `unchanged` - the collection of items that have not changed on the attribute (the second tuple element).
- `deleted` - the collection of items that have been removed from the attribute (the third tuple element).

empty()

Return True if this `History` has no changes and no existing, unchanged state.

has_changes()

Return True if this `History` has changes.

non_added()

Return a collection of unchanged + deleted.

non_deleted()

Return a collection of added + unchanged.

sum()

Return a collection of added + unchanged + deleted.

Previous: Tracking Object and Session Changes with Events Next: Events and Internals