Column and Data Types — SQLAlchemy 1.3 Documentation

SQLAlchemy provides abstractions for most common database data types, and a mechanism for specifying your own custom data types.

The methods and attributes of type objects are rarely used directly. Type objects are supplied to `Table` definitions and can be supplied as type hints to *functions* for occasions where the database driver returns an incorrect type.

SQLAlchemy will use the `Integer` and `String(32)` type information when issuing a `CREATE TABLE` statement and will use it again when reading back rows `SELECTed` from the database. Functions that accept a type (such as `Column()`) will typically accept a type class or instance; `Integer` is equivalent to `Integer()` with no construction arguments in this case.

Generic Types¶

Generic types specify a column that can read, write and store a particular type of Python data. SQLAlchemy will choose the best database column type available on the target database when issuing a `CREATE TABLE` statement. For complete control over which column type is emitted in `CREATE TABLE`, such as `VARCHAR` see SQL Standard and Multiple Vendor Types and the other sections of this chapter.

class `sqlalchemy.types.BigInteger`¶

Bases: `sqlalchemy.types.Integer`

A type for bigger `int` integers.

Typically generates a `BIGINT` in DDL, and otherwise acts like a normal `Integer` on the Python side.

class `sqlalchemy.types.Boolean`(*create_constraint=True*, *name=None*, *_create_events=True*)¶

Bases: `sqlalchemy.types.Emulated`, `sqlalchemy.types.TypeEngine`, `sqlalchemy.types.SchemaType`

A bool datatype.

`Boolean` typically uses BOOLEAN or SMALLINT on the DDL side, and on the Python side deals in `True` or `False`.

The `Boolean` datatype currently has two levels of assertion that the values persisted are simple true/false values. For all backends, only the Python values `None`, `True`, `False`, `1` or `0` are accepted as parameter values. For those backends that don't support a "native boolean" datatype, a CHECK constraint is also created on the target column. Production of the CHECK constraint can be

disabled by passing the `Boolean.create_constraint` flag set to `False`.

Changed in version 1.2: the `Boolean` datatype now asserts that incoming Python values are already in pure boolean form.

`__init__`(*create_constraint=True*, *name=None*, *_create_events=True*)¶

Construct a Boolean.

Parameters

- **create_constraint**¶ – defaults to True. If the boolean is generated as an int/smallint, also create a CHECK constraint on the table that ensures 1 or 0 as a value.
- **name**¶ – if a CHECK constraint is generated, specify the name of the constraint.

`bind_processor`(*dialect*)¶

Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

Parameters

dialect¶ – Dialect instance in use.

`literal_processor`(*dialect*)¶

Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the “literal_binds” flag, typically used in DDL generation as well as in certain scenarios where backends don’t accept bound parameters.

New in version 0.9.0.

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`result_processor`(*dialect*, *coltype*)¶

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect**¶ – Dialect instance in use.
- **coltype**¶ – DBAPI coltype argument received in cursor.description.

class `sqlalchemy.types.Date`¶

Bases: `sqlalchemy.types._LookupExpressionAdapter`, `sqlalchemy.types.TypeEngine`

A type for `datetime.date()` objects.

`get_dbapi_type`(*dbapi*)¶

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

class `sqlalchemy.types.DateTime`(*timezone=False*)¶

Bases: `sqlalchemy.types._LookupExpressionAdapter`, `sqlalchemy.types.TypeEngine`

A type for `datetime.datetime()` objects.

Date and time types return objects from the Python `datetime` module. Most DBAPIs have built in support for the datetime module, with the noted exception of SQLite. In the case of SQLite, date and time types are stored as strings which are then converted back to datetime objects when rows are returned.

For the time representation within the datetime type, some backends include additional options, such as timezone support and fractional seconds support. For fractional seconds, use the dialect-specific datatype, such as `mysql.TIME`. For timezone support, use at least the `TIMESTAMP` datatype, if not the dialect-specific datatype object.

`__init__`(*timezone=False*)¶

Construct a new `DateTime`.

Parameters

timezone¶ – boolean. Indicates that the datetime type should enable timezone support, if available on the **base date/time-holding type only**. It is recommended to make use of the `TIMESTAMP` datatype directly when using this flag, as some databases include separate generic date/time-holding types distinct from the timezone-capable TIMESTAMP datatype, such as Oracle.

`get_dbapi_type`(*dbapi*)¶

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

class `sqlalchemy.types.Enum`(**enums*, ***kw*)¶

Bases: `sqlalchemy.types.Emulated`, `sqlalchemy.types.String`, `sqlalchemy.types.SchemaType`

Generic Enum Type.

The `Enum` type provides a set of possible string values which the column is constrained towards.

The `Enum` type will make use of the backend's native "ENUM" type if one is available; otherwise, it uses a VARCHAR datatype and produces a CHECK constraint. Use of the backend-native enum type can be disabled using the `Enum.native_enum` flag, and the production of the CHECK constraint is configurable using the `Enum.create_constraint` flag.

The Enum type also provides in-Python validation of string values during both read and write operations. When reading a value from the database in a result set, the string value is always checked against the list of possible values and a LookupError is raised if no match is found. When passing a value to the database as a plain string within a SQL statement, if the Enum.validate_strings parameter is set to True, a LookupError is raised for any string value that's not located in the given list of possible values; note that this impacts usage of LIKE expressions with enumerated values (an unusual use case).

Changed in version 1.1: the Enum type now provides in-Python validation of input values as well as on data being returned by the database.

The source of enumerated values may be a list of string values, or alternatively a PEP-435-compliant enumerated class. For the purposes of the Enum datatype, this class need only provide a __members__ method.

When using an enumerated class, the enumerated objects are used both for input and output, rather than strings as is the case with a plain-string enumerated type:

```
import enum
class MyEnum(enum.Enum):
    one = 1
    two = 2
    three = 3

t = Table(
    'data', MetaData(),
    Column('value', Enum(MyEnum))
)

connection.execute(t.insert(), {"value": MyEnum.two})
assert connection.scalar(t.select()) is MyEnum.two
```

Above, the string names of each element, e.g. "one", "two", "three", are persisted to the database; the values of the Python Enum, here indicated as integers, are **not** used; the value of each enum can therefore be any kind of Python object whether or not it is persistable.

In order to persist the values and not the names, the Enum.values_callable parameter may be used. The value of this parameter is a user-supplied callable, which is intended to be used with a PEP-435-compliant enumerated class and returns a list of string values to be persisted. For a simple enumeration that uses string values, a callable such as lambda x: [e.value for e in x] is sufficient.

New in version 1.1: - support for PEP-435-style enumerated classes.

__init__(*enums*, ***kw*)¶

Construct an enum.

Keyword arguments which don't apply to a specific backend are ignored by that backend.

Parameters

- ***enums**¶ –

  either exactly one PEP-435 compliant enumerated type or one or more string or unicode enumeration labels. If unicode labels are present, the *convert_unicode* flag is auto-enabled.

  New in version 1.1: a PEP-435 style enumerated class may be passed.

- **convert_unicode**¶ –

  Enable unicode-aware bind parameter and result-set processing for this Enum's data. This is set automatically based on the presence of unicode label strings.

  Deprecated since version 1.3: The `Enum.convert_unicode` parameter is deprecated and will be removed in a future release. All modern DBAPIs now support Python Unicode directly and this parameter is unnecessary.

- **create_constraint**¶ –

  defaults to True. When creating a non-native enumerated type, also build a CHECK constraint on the database against the valid values.

  New in version 1.1: - added `Enum.create_constraint` which provides the option to disable the production of the CHECK constraint for a non-native enumerated type.

- **metadata**¶ – Associate this type directly with a `MetaData` object. For types that exist on the target database as an independent schema construct (PostgreSQL), this type will be created and dropped within `create_all()` and `drop_all()` operations. If the type is not associated with any `MetaData` object, it will associate itself with each `Table` in which it is used, and will be created when any of those individual tables are created, after a check is performed for its existence. The type is only dropped when `drop_all()` is called for that `Table` object's metadata, however.

- **name**¶ – The name of this type. This is required for PostgreSQL and any future supported database which requires an explicitly named type, or an explicitly named constraint in order to generate the type and/or a table that uses it. If a PEP-435 enumerated class was used, its name (converted to lower case) is used by default.

- **native_enum**¶ – Use the database's native ENUM type when available. Defaults to True. When False, uses VARCHAR + check constraint for all backends.

- **schema**¶ –

Schema name of this type. For types that exist on the target database as an independent schema construct (PostgreSQL), this parameter specifies the named schema in which the type is present.

Note

The `schema` of the `Enum` type does not by default make use of the `schema` established on the owning `Table`. If this behavior is desired, set the `inherit_schema` flag to `True`.

- **quote**¶ – Set explicit quoting preferences for the type's name.
- **inherit_schema**¶ – When `True`, the "schema" from the owning `Table` will be copied to the "schema" attribute of this `Enum`, replacing whatever value was passed for the `schema` attribute. This also takes effect when using the `Table.tometadata()` operation.
- **validate_strings**¶ –

  when True, string values that are being passed to the database in a SQL statement will be checked for validity against the list of enumerated values. Unrecognized values will result in a `LookupError` being raised.

  New in version 1.1.0b2.
- **values_callable**¶ –

  A callable which will be passed the PEP-435 compliant enumerated type, which should then return a list of string values to be persisted. This allows for alternate usages such as using the string value of an enum to be persisted to the database instead of its name.

  New in version 1.2.3.
- **sort_key_function**¶ –

  a Python callable which may be used as the "key" argument in the Python `sorted()` built-in. The SQLAlchemy ORM requires that primary key columns which are mapped must be sortable in some way. When using an unsortable enumeration object such as a Python 3 `Enum` object, this parameter may be used to set a default sort key function for the objects. By default, the database value of the enumeration is used as the sorting function.

`create`(*bind=None*, *checkfirst=False*)¶

Issue CREATE ddl for this type, if applicable.

`drop`(*bind=None*, *checkfirst=False*)¶

Issue DROP ddl for this type, if applicable.

class `sqlalchemy.types.Float`(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*)¶

Bases: `sqlalchemy.types.Numeric`

Type representing floating point types, such as `FLOAT` or `REAL`.

This type returns Python `float` objects by default, unless the `Float.asdecimal` flag is set to True, in which case they are coerced to `decimal.Decimal` objects.

Note

The `Float` type is designed to receive data from a database type that is explicitly known to be a floating point type (e.g. `FLOAT`, `REAL`, others) and not a decimal type (e.g. `DECIMAL`, `NUMERIC`, others). If the database column on the server is in fact a Numeric type, such as `DECIMAL` or `NUMERIC`, use the `Numeric` type or a subclass, otherwise numeric coercion between `float`/`Decimal` may or may not function as expected.

`__init__`(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*)¶

Construct a Float.

Parameters

- **precision**¶ – the numeric precision for use in DDL `CREATE TABLE`.
- **asdecimal**¶ – the same flag as that of `Numeric`, but defaults to `False`. Note that setting this flag to `True` results in floating point conversion.
- **decimal_return_scale**¶ –

  Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specifying this value will override that length. Note that the MySQL float types, which do include "scale", will use "scale" as the default for decimal_return_scale, if not otherwise specified.

  New in version 0.9.0.

`result_processor`(*dialect*, *coltype*)¶

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect**¶ – Dialect instance in use.
- **coltype**¶ – DBAPI coltype argument received in cursor.description.

class `sqlalchemy.types.Integer`¶

Bases: `sqlalchemy.types._LookupExpressionAdapter`, `sqlalchemy.types.TypeEngine`

A type for `int` integers.

`get_dbapi_type`(*dbapi*)¶

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

`literal_processor`(*dialect*)¶

Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the “literal_binds” flag, typically used in DDL generation as well as in certain scenarios where backends don’t accept bound parameters.

New in version 0.9.0.

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

class `sqlalchemy.types.Interval`(*native=True*, *second_precision=None*, *day_precision=None*)¶

Bases: `sqlalchemy.types.Emulated`, `sqlalchemy.types._AbstractInterval`, `sqlalchemy.types.TypeDecorator`

A type for `datetime.timedelta()` objects.

The Interval type deals with `datetime.timedelta` objects. In PostgreSQL, the native `INTERVAL` type is used; for others, the value is stored as a date which is relative to the "epoch" (Jan. 1, 1970).

Note that the `Interval` type does not currently provide date arithmetic operations on platforms which do not support interval types natively. Such operations usually require transformation of both sides of the expression (such as, conversion of both sides into integer epoch values first) which currently is a manual procedure (such as via `func`).

`__init__`(*native=True*, *second_precision=None*, *day_precision=None*)¶

Construct an Interval object.

Parameters

- **native**¶ – when True, use the actual INTERVAL type provided by the database, if supported (currently PostgreSQL, Oracle). Otherwise, represent the interval data as an epoch value regardless.
- **second_precision**¶ – For native interval types which support a "fractional seconds precision" parameter, i.e. Oracle and PostgreSQL
- **day_precision**¶ – for native interval types which support a "day precision" parameter, i.e. Oracle.

`adapt_to_emulated`(*impltype*, ***kw*)¶

Given an impl class, adapt this type to the impl assuming "emulated".

The impl should also be an "emulated" version of this type, most likely the same class as this type itself.

e.g.: sqltypes.Enum adapts to the Enum class.

`bind_processor`(*dialect*)¶

Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

Parameters

dialect¶ – Dialect instance in use.

`impl`¶

alias of `DateTime`

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`result_processor`(*dialect*, *coltype*)¶

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect**¶ – Dialect instance in use.
- **coltype**¶ – DBAPI coltype argument received in cursor.description.

class `sqlalchemy.types.LargeBinary`(*length=None*)¶

Bases: `sqlalchemy.types._Binary`

A type for large binary byte data.

The `LargeBinary` type corresponds to a large and/or unlengthed binary type for the target platform, such as BLOB on MySQL and BYTEA for PostgreSQL. It also handles the necessary conversions for the DBAPI.

`__init__`(*length=None*)¶

Construct a LargeBinary type.

Parameters

length¶ – optional, a length for the column for use in DDL statements, for those binary types that accept a length, such as the MySQL BLOB type.

class `sqlalchemy.types.MatchType`(*create_constraint=True*, *name=None*, *_create_events=True*)¶

Bases: `sqlalchemy.types.Boolean`

Refers to the return type of the MATCH operator.

As the ColumnOperators.match() is probably the most open-ended operator in generic SQLAlchemy Core, we can't assume the return type at SQL evaluation time, as MySQL returns a floating point, not a boolean, and other backends might do something different. So this type acts as a placeholder, currently subclassing Boolean. The type allows dialects to inject result-processing functionality if needed, and on MySQL will return floating-point values.

New in version 1.0.0.

class sqlalchemy.types.Numeric(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)¶

Bases: sqlalchemy.types._LookupExpressionAdapter, sqlalchemy.types.TypeEngine

A type for fixed precision numbers, such as NUMERIC or DECIMAL.

This type returns Python decimal.Decimal objects by default, unless the Numeric.asdecimal flag is set to False, in which case they are coerced to Python float objects.

Note

The Numeric type is designed to receive data from a database type that is explicitly known to be a decimal type (e.g. DECIMAL, NUMERIC, others) and not a floating point type (e.g. FLOAT, REAL, others). If the database column on the server is in fact a floating-point type type, such as FLOAT or REAL, use the Float type or a subclass, otherwise numeric coercion between float/Decimal may or may not function as expected.

Note

The Python decimal.Decimal class is generally slow performing; cPython 3.3 has now switched to use the cdecimal library natively. For older Python versions, the cdecimal library can be patched into any application where it will replace the decimal library fully, however this needs to be applied globally and before any other modules have been imported, as follows:

```
import sys
import cdecimal
sys.modules["decimal"] = cdecimal
```

Note that the cdecimal and decimal libraries are **not compatible with each other**, so patching cdecimal at the global level is the only way it can be used effectively with various DBAPIs that hardcode to import the decimal library.

__init__(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)¶

Construct a Numeric.

Parameters

- **precision**¶ – the numeric precision for use in DDL CREATE

TABLE.

- **scale**¶ – the numeric scale for use in DDL CREATE TABLE.
- **asdecimal**¶ – default True. Return whether or not values should be sent as Python Decimal objects, or as floats. Different DBAPIs send one or the other based on datatypes - the Numeric type will ensure that return values are one or the other across DBAPIs consistently.
- **decimal_return_scale**¶ –

  Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specifying this value will override that length. Types which do include an explicit ".scale" value, such as the base Numeric as well as the MySQL float types, will use the value of ".scale" as the default for decimal_return_scale, if not otherwise specified.

  New in version 0.9.0.

When using the Numeric type, care should be taken to ensure that the asdecimal setting is appropriate for the DBAPI in use - when Numeric applies a conversion from Decimal->float or float-> Decimal, this conversion incurs an additional performance overhead for all result columns received.

DBAPIs that return Decimal natively (e.g. psycopg2) will have better accuracy and higher performance with a setting of True, as the native translation to Decimal reduces the amount of floating- point issues at play, and the Numeric type itself doesn't need to apply any further conversions. However, another DBAPI which returns floats natively *will* incur an additional conversion overhead, and is still subject to floating point data loss - in which case asdecimal=False will at least remove the extra conversion overhead.

bind_processor(*dialect*)¶

Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return None.

Parameters

dialect¶ – Dialect instance in use.

get_dbapi_type(*dbapi*)¶

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

`literal_processor`(*dialect*)¶

Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the "literal_binds" flag, typically used in DDL generation as well as in certain scenarios where backends don't accept bound parameters.

New in version 0.9.0.

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`result_processor`(*dialect*, *coltype*)¶

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect**¶ – Dialect instance in use.
- **coltype**¶ – DBAPI coltype argument received in cursor.description.

class `sqlalchemy.types.PickleType`(*protocol=4*, *pickler=None*, *comparator=None*)¶

Bases: `sqlalchemy.types.TypeDecorator`

Holds Python objects, which are serialized using pickle.

PickleType builds upon the Binary type to apply Python's `pickle.dumps()` to incoming objects, and `pickle.loads()` on the way out, allowing any pickleable Python object to be stored as a serialized binary field.

To allow ORM change events to propagate for elements associated with PickleType, see Mutation Tracking.

__init__(*protocol=4*, *pickler=None*, *comparator=None*)¶

Construct a PickleType.

Parameters

- **protocol**¶ – defaults to pickle.HIGHEST_PROTOCOL.
- **pickler**¶ – defaults to cPickle.pickle or pickle.pickle if cPickle is not available. May be any object with pickle-compatible dumps` and ``loads methods.
- **comparator**¶ – a 2-arg callable predicate used to compare values of this type. If left as None, the Python "equals" operator is used to compare values.

bind_processor(*dialect*)¶

Provide a bound value processing function for the given Dialect.

This is the method that fulfills the TypeEngine contract for bound value conversion. TypeDecorator will wrap a user-defined implementation of process_bind_param() here.

User-defined code can override this method directly, though its likely best to use process_bind_param() so that the processing provided by self.impl is maintained.

Parameters

dialect¶ – Dialect instance in use.

This method is the reverse counterpart to the result_processor() method of this class.

compare_values(*x*, *y*)¶

Given two values, compare them for equality.

By default this calls upon TypeEngine.compare_values() of the underlying "impl", which in turn usually uses the Python equals operator ==.

This function is used by the ORM to compare an original-loaded value with an intercepted "changed" value, to determine if a net change has occurred.

impl¶

alias of LargeBinary

result_processor(*dialect*, *coltype*)¶

Provide a result value processing function for the given Dialect.

This is the method that fulfills the TypeEngine contract for result value conversion. TypeDecorator will wrap a user-defined implementation of process_result_value() here.

User-defined code can override this method directly, though its likely best to use process_result_value() so that the processing provided by self.impl is maintained.

Parameters

- **dialect¶** – Dialect instance in use.
- **coltype¶** – A SQLAlchemy data type

This method is the reverse counterpart to the bind_processor() method of this class.

class sqlalchemy.types.SchemaType(*name=None*, *schema=None*, *metadata=None*, *inherit_schema=False*, *quote=None*, *_create_events=True*)¶

Bases: sqlalchemy.sql.expression.SchemaEventTarget

Mark a type as possibly requiring schema-level DDL for usage.

Supports types that must be explicitly created/dropped (i.e. PG ENUM type) as well as types that are complimented by table or schema level constraints, triggers, and other rules.

SchemaType classes can also be targets for the DDLEvents.before_parent_attach() and DDLEvents.after_parent_attach() events, where the events fire off surrounding the association of the type object with a parent Column.

adapt(*impltype*, ***kw*)¶
bind¶
copy(***kw*)¶
create(*bind=None*, *checkfirst=False*)¶

Issue CREATE ddl for this type, if applicable.

drop(*bind=None*, *checkfirst=False*)¶

Issue DROP ddl for this type, if applicable.

class sqlalchemy.types.SmallInteger¶

Bases: sqlalchemy.types.Integer

A type for smaller int integers.

Typically generates a SMALLINT in DDL, and otherwise acts like a normal Integer on the Python side.

class `sqlalchemy.types.String`(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*, *_expect_unicode=False*)¶

Bases: `sqlalchemy.types.Concatenable`, `sqlalchemy.types.TypeEngine`

The base for all string and character types.

In SQL, corresponds to VARCHAR. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.)

The *length* field is usually required when the *String* type is used within a CREATE TABLE statement, as VARCHAR requires a length on most databases.

`__init__`(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*, *_expect_unicode=False*)¶

Create a string-holding type.

Parameters

- **length**¶ – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a `length` for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **collation**¶ –

  Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:

  ```
  >>> from sqlalchemy import cast, select, String
  >>> print select([cast('some string', String(collation='utf8'))])
  SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
  ```

- **convert_unicode**¶ –

  When set to `True`, the `String` type will assume that input is to be passed as Python Unicode objects under Python 2, and results returned as Python Unicode objects. In the rare circumstance that the DBAPI does not support Python unicode under Python 2, SQLAlchemy will use its own encoder/decoder functionality on strings, referring to the value of the `create_engine.encoding` parameter parameter passed to `create_engine()` as the encoding.

  Deprecated since version 1.3: The `String.convert_unicode` parameter is deprecated and will be removed in a future release. All modern DBAPIs now support Python Unicode

directly and this parameter is unnecessary.

For the extremely rare case that Python Unicode is to be encoded/decoded by SQLAlchemy on a backend that *does* natively support Python Unicode, the string value `"force"` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

Note

SQLAlchemy's unicode-conversion flags and features only apply to Python 2; in Python 3, all string objects are Unicode objects. For this reason, as well as the fact that virtually all modern DBAPIs now support Unicode natively even under Python 2, the `String.convert_unicode` flag is inherently a legacy feature.

Note

In the vast majority of cases, the `Unicode` or `UnicodeText` datatypes should be used for a `Column` that expects to store non-ascii data. These datatypes will ensure that the correct types are used on the database side as well as set up the correct Unicode behaviors under Python 2.

- **unicode_error**¶ –

  Optional, a method to use to handle Unicode conversion errors. Behaves like the `errors` keyword argument to the standard library's `string.decode()` functions, requires that `String.convert_unicode` is set to `"force"`

  Deprecated since version 1.3: The `String.unicode_errors` parameter is deprecated and will be removed in a future release. This parameter is unnecessary for modern Python DBAPIs and degrades performance significantly.

`bind_processor`(*dialect*)¶

Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

Parameters

dialect¶ – Dialect instance in use.

`get_dbapi_type`(*dbapi*)¶

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

`literal_processor`(*dialect*)¶

Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the “literal_binds” flag, typically used in DDL generation as well as in certain scenarios where backends don’t accept bound parameters.

New in version 0.9.0.

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`result_processor`(*dialect*, *coltype*)¶

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect**¶ – Dialect instance in use.
- **coltype**¶ – DBAPI coltype argument received in cursor.description.

class `sqlalchemy.types.Text`(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*, *_expect_unicode=False*)¶

Bases: `sqlalchemy.types.String`

A variably sized string type.

In SQL, usually corresponds to CLOB or TEXT. Can also take Python unicode objects and encode to the database’s encoding in bind params (and the reverse for result sets.) In general, TEXT objects do not have a length; while some databases will accept a length argument here, it will be rejected by others.

class sqlalchemy.types.Time(*timezone=False*)¶

Bases: sqlalchemy.types._LookupExpressionAdapter, sqlalchemy.types.TypeEngine

A type for datetime.time() objects.

get_dbapi_type(*dbapi*)¶

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling setinputsizes(), for example.

python_type¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like int for example), will return that type.

If a return type is not defined, raises NotImplementedError.

Note that any type also accommodates NULL in SQL which means you can also get back None from any type in practice.

class sqlalchemy.types.Unicode(*length=None*, ***kwargs*)¶

Bases: sqlalchemy.types.String

A variable length Unicode string type.

The Unicode type is a String subclass that assumes input and output as Python unicode data, and in that regard is equivalent to the usage of the convert_unicode flag with the String type. However, unlike plain String, it also implies an underlying column type that is explicitly supporting of non-ASCII data, such as NVARCHAR on Oracle and SQL Server. This can impact the output of CREATE TABLE statements and CAST functions at the dialect level, and can also affect the handling of bound parameters in some specific DBAPI scenarios.

The encoding used by the Unicode type is usually determined by the DBAPI itself; most modern DBAPIs feature support for Python unicode objects as bound values and result set values, and the encoding should be configured as detailed in the notes for the target DBAPI in the Dialects section.

For those DBAPIs which do not support, or are not configured to accommodate Python unicode objects directly, SQLAlchemy does the encoding and decoding outside of the DBAPI. The encoding in this scenario is determined by the encoding flag passed to create_engine().

When using the Unicode type, it is only appropriate to pass Python unicode objects, and not plain str. If a plain str is passed under Python 2, a warning is emitted. If you notice your application emitting these warnings but you're not

sure of the source of them, the Python `warnings` filter, documented at http://docs.python.org/library/warnings.html, can be used to turn these warnings into exceptions which will illustrate a stack trace:

```
import warnings
warnings.simplefilter('error')
```

For an application that wishes to pass plain bytestrings and Python `unicode` objects to the `Unicode` type equally, the bytestrings must first be decoded into unicode. The recipe at Coercing Encoded Strings to Unicode illustrates how this is done.

`__init__`(*length=None*, ***kwargs*)¶

Create a `Unicode` object.

Parameters are the same as that of `String`, with the exception that `convert_unicode` defaults to `True`.

class `sqlalchemy.types.UnicodeText`(*length=None*, ***kwargs*)¶

Bases: `sqlalchemy.types.Text`

An unbounded-length Unicode string type.

See `Unicode` for details on the unicode behavior of this object.

Like `Unicode`, usage the `UnicodeText` type implies a unicode-capable type being used on the backend, such as `NCLOB`, `NTEXT`.

`__init__`(*length=None*, ***kwargs*)¶

Create a Unicode-converting Text type.

Parameters are the same as that of `Text`, with the exception that `convert_unicode` defaults to `True`.

SQL Standard and Multiple Vendor Types¶

This category of types refers to types that are either part of the SQL standard, or are potentially found within a subset of database backends. Unlike the “generic” types, the SQL standard/multi-vendor types have **no** guarantee of working on all backends, and will only work on those backends that explicitly support them by name. That is, the type will always emit its exact name in DDL with `CREATE TABLE` is issued.

class `sqlalchemy.types.ARRAY`(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)¶

Bases: `sqlalchemy.sql.expression.SchemaEventTarget`, `sqlalchemy.types.Indexable`, `sqlalchemy.types.Concatenable`, `sqlalchemy.types.TypeEngine`

Represent a SQL Array type.

Note

This type serves as the basis for all ARRAY operations. However, currently **only the PostgreSQL backend has support for SQL arrays in SQLAlchemy**. It is recommended to use the postgresql.ARRAY type directly when using ARRAY types with PostgreSQL, as it provides additional operators specific to that backend.

types.ARRAY is part of the Core in support of various SQL standard functions such as array_agg which explicitly involve arrays; however, with the exception of the PostgreSQL backend and possibly some third-party dialects, no other SQLAlchemy built-in dialect has support for this type.

An types.ARRAY type is constructed given the "type" of element:

```
mytable = Table("mytable", metadata,
        Column("data", ARRAY(Integer))
    )
```

The above type represents an N-dimensional array, meaning a supporting backend such as PostgreSQL will interpret values with any number of dimensions automatically. To produce an INSERT construct that passes in a 1-dimensional array of integers:

```
connection.execute(
        mytable.insert(),
        data=[1,2,3]
)
```

The types.ARRAY type can be constructed given a fixed number of dimensions:

```
mytable = Table("mytable", metadata,
        Column("data", ARRAY(Integer, dimensions=2))
    )
```

Sending a number of dimensions is optional, but recommended if the datatype is to represent arrays of more than one dimension. This number is used:

- When emitting the type declaration itself to the database, e.g. `INTEGER[][]`
- When translating Python values to database values, and vice versa, e.g. an ARRAY of Unicode objects uses this number to efficiently access the string values inside of array structures without resorting to per-row type inspection
- When used with the Python `getitem` accessor, the number of dimensions serves to define the kind of type that the `[]` operator should return, e.g. for an ARRAY of INTEGER with two dimensions:

  ```
  >>> expr = table.c.column[5]  # returns ARRAY(Integer, dimensions=1)
  >>> expr = expr[6]  # returns Integer
  ```

For 1-dimensional arrays, an types.ARRAY instance with no dimension parameter will generally assume single-dimensional behaviors.

SQL expressions of type types.ARRAY have support for "index" and "slice"

behavior. The Python `[]` operator works normally here, given integer indexes or slices. Arrays default to 1-based indexing. The operator produces binary expression constructs which will produce the appropriate SQL, both for SELECT statements:

```
select([mytable.c.data[5], mytable.c.data[2:7]])
```

as well as UPDATE statements when the `Update.values()` method is used:

```
mytable.update().values({
    mytable.c.data[5]: 7,
    mytable.c.data[2:7]: [1, 2, 3]
})
```

The `types.ARRAY` type also provides for the operators `types.ARRAY.Comparator.any()` and `types.ARRAY.Comparator.all()`. The PostgreSQL-specific version of `types.ARRAY` also provides additional operators.

New in version 1.1.0.

class `Comparator`(*expr*)¶

Bases: `sqlalchemy.types.Comparator`, `sqlalchemy.types.Comparator`

Define comparison operations for `types.ARRAY`.

More operators are available on the dialect-specific form of this type. See `postgresql.ARRAY.Comparator`.

`all`(*elements*, *other*, *operator=None*)¶

Return `other operator ALL (array)` clause.

Argument places are switched, because ALL requires array expression to be on the right hand-side.

E.g.:

```
from sqlalchemy.sql import operators

conn.execute(
    select([table.c.data]).where(
            table.c.data.all(7, operator=operators.lt)
        )
)
```

Parameters

- **other**¶ – expression to be compared
- **operator**¶ – an operator object from the `sqlalchemy.sql.operators` package, defaults to `operators.eq()`.

`any`(*elements*, *other*, *operator=None*)¶

Return `other operator ANY (array)` clause.

Argument places are switched, because ANY requires array expression to be on the right hand-side.

E.g.:

```
from sqlalchemy.sql import operators

conn.execute(
    select([table.c.data]).where(
            table.c.data.any(7, operator=operators.lt)
        )
)
```

Parameters

- **other¶** – expression to be compared
- **operator¶** – an operator object from the `sqlalchemy.sql.operators` package, defaults to `operators.eq()`.

`contains`(**arg*, ***kw*)¶

Implement the 'contains' operator.

Produces a LIKE expression that tests against a match for the middle of a string value:

```
column LIKE '%' || <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.contains("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.contains.autoescape` flag may be set to `True` to apply escaping to occurrences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.contains.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other¶** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and _ are not escaped by default unless the `ColumnOperators.contains.autoescape` flag is set to True.
- **autoescape¶** –

  boolean; when True, establishes an escape character

within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

An expression such as:

```
somecolumn.contains("foo%bar", autoescape=True)
```

Will render as:

```
somecolumn LIKE '%' || :param || '%' ESCAPE '/'
```

With the value of :param as `"foo/%bar"`.

New in version 1.2.

Changed in version 1.2.0: The `ColumnOperators.contains.autoescape` parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the `ColumnOperators.contains.escape` parameter.

- **escape**¶ –

  a character which when given will render with the `ESCAPE` keyword to establish that character as the escape character. This character can then be placed preceding occurrences of `%` and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.contains("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '^'
  ```

  The parameter may also be combined with `ColumnOperators.contains.autoescape`:

  ```
  somecolumn.contains("foo%bar^bat", escape="^", autoescape=T
  ```

  Where above, the given literal parameter will be converted to `"foo^%bar^^bat"` before being passed to the database.

__init__(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)¶

Construct an `types.ARRAY`.

E.g.:

```
Column('myarray', ARRAY(Integer))
```

Arguments are:

Parameters

- **item_type¶** – The data type of items of this array. Note that dimensionality is irrelevant here, so multi-dimensional arrays like `INTEGER[][]`, are constructed as `ARRAY(Integer)`, not as `ARRAY(ARRAY(Integer))` or such.
- **as_tuple=False¶** – Specify whether return results should be converted to tuples from lists. This parameter is not generally needed as a Python list corresponds well to a SQL array.
- **dimensions¶** – if non-None, the ARRAY will assume a fixed number of dimensions. This impacts how the array is declared on the database, how it goes about interpreting Python and result values, as well as how expression behavior in conjunction with the "getitem" operator works. See the description at `types.ARRAY` for additional detail.
- **zero_indexes=False¶** – when True, index values will be converted between Python zero-based and SQL one-based indexes, e.g. a value of one will be added to all index values before passing to the database.

`comparator_factory`¶

alias of `ARRAY.Comparator`

`compare_values`(*x*, *y*)¶

Compare two values for equality.

`hashable`¶

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you

can also get back `None` from any type in practice.

`zero_indexes` = *False*¶

if True, Python zero-based indexes should be interpreted as one-based on the SQL expression side.

class `sqlalchemy.types.BIGINT`¶

Bases: `sqlalchemy.types.BigInteger`

The SQL BIGINT type.

class `sqlalchemy.types.BINARY`(*length=None*)¶

Bases: `sqlalchemy.types._Binary`

The SQL BINARY type.

class `sqlalchemy.types.BLOB`(*length=None*)¶

Bases: `sqlalchemy.types.LargeBinary`

The SQL BLOB type.

class `sqlalchemy.types.BOOLEAN`(*create_constraint=True*, *name=None*, *_create_events=True*)¶

Bases: `sqlalchemy.types.Boolean`

The SQL BOOLEAN type.

class `sqlalchemy.types.CHAR`(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*, *_expect_unicode=False*)¶

Bases: `sqlalchemy.types.String`

The SQL CHAR type.

class `sqlalchemy.types.CLOB`(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*, *_expect_unicode=False*)¶

Bases: `sqlalchemy.types.Text`

The CLOB type.

This type is found in Oracle and Informix.

class `sqlalchemy.types.DATE`¶

Bases: `sqlalchemy.types.Date`

The SQL DATE type.

class `sqlalchemy.types.DATETIME`(*timezone=False*)¶

Bases: sqlalchemy.types.DateTime

The SQL DATETIME type.

class sqlalchemy.types.DECIMAL(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)¶

Bases: sqlalchemy.types.Numeric

The SQL DECIMAL type.

class sqlalchemy.types.FLOAT(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*)¶

Bases: sqlalchemy.types.Float

The SQL FLOAT type.

sqlalchemy.types.INT¶

alias of sqlalchemy.sql.sqltypes.INTEGER

class sqlalchemy.types.JSON(*none_as_null=False*)¶

Bases: sqlalchemy.types.Indexable, sqlalchemy.types.TypeEngine

Represent a SQL JSON type.

Note

types.JSON is provided as a facade for vendor-specific JSON types. Since it supports JSON SQL operations, it only works on backends that have an actual JSON type, currently:

- PostgreSQL
- MySQL as of version 5.7 (MariaDB as of the 10.2 series does not)
- SQLite as of version 3.9

types.JSON is part of the Core in support of the growing popularity of native JSON datatypes.

The types.JSON type stores arbitrary JSON format data, e.g.:

```
data_table = Table('data_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', JSON)
)

with engine.connect() as conn:
    conn.execute(
        data_table.insert(),
        data = {"key1": "value1", "key2": "value2"}
    )
```

JSON-Specific Expression Operators

The types.JSON datatype provides these additional SQL operations:

- Keyed index operations:

  ```
  data_table.c.data['some key']
  ```

- Integer index operations:

- Path index operations:

  ```
  data_table.c.data[('key_1', 'key_2', 5, ..., 'key_n')]
  ```

- Data casters for specific JSON element types, subsequent to an index or path operation being invoked:

  ```
  data_table.c.data["some key"].as_integer()
  ```

  New in version 1.3.11.

Additional operations may be available from the dialect-specific versions of types.JSON, such as postgresql.JSON and postgresql.JSONB which both offer additional PostgreSQL-specific operations.

Casting JSON Elements to Other Types

Index operations, i.e. those invoked by calling upon the expression using the Python bracket operator as in `some_column['some key']`, return an expression object whose type defaults to JSON by default, so that further JSON-oriented instructions may be called upon the result type. However, it is likely more common that an index operation is expected to return a specific scalar element, such as a string or integer. In order to provide access to these elements in a backend-agnostic way, a series of data casters are provided:

- JSON.Comparator.as_string() - return the element as a string
- JSON.Comparator.as_boolean() - return the element as a boolean
- JSON.Comparator.as_float() - return the element as a float
- JSON.Comparator.as_integer() - return the element as an integer

These data casters are implemented by supporting dialects in order to assure that comparisons to the above types will work as expected, such as:

```
# integer comparison
data_table.c.data["some_integer_key"].as_integer() == 5

# boolean comparison
data_table.c.data["some_boolean"].as_boolean() == True
```

New in version 1.3.11: Added type-specific casters for the basic JSON data element types.

Note

The data caster functions are new in version 1.3.11, and supersede the previous

documented approaches of using CAST; for reference, this looked like:

```
from sqlalchemy import cast, type_coerce
from sqlalchemy import String, JSON
cast(
    data_table.c.data['some_key'], String
) == type_coerce(55, JSON)
```

The above case now works directly as:

```
data_table.c.data['some_key'].as_integer() == 5
```

For details on the previous comparison approach within the 1.3.x series, see the documentation for SQLAlchemy 1.2 or the included HTML files in the doc/ directory of the version's distribution.

Detecting Changes in JSON columns when using the ORM

The JSON type, when used with the SQLAlchemy ORM, does not detect in-place mutations to the structure. In order to detect these, the sqlalchemy.ext.mutable extension must be used. This extension will allow "in-place" changes to the datastructure to produce events which will be detected by the unit of work. See the example at HSTORE for a simple example involving a dictionary.

Support for JSON null vs. SQL NULL

When working with NULL values, the JSON type recommends the use of two specific constants in order to differentiate between a column that evaluates to SQL NULL, e.g. no value, vs. the JSON-encoded string of `"null"`. To insert or select against a value that is SQL NULL, use the constant null():

```
from sqlalchemy import null
conn.execute(table.insert(), json_value=null())
```

To insert or select against a value that is JSON `"null"`, use the constant JSON.NULL:

```
conn.execute(table.insert(), json_value=JSON.NULL)
```

The JSON type supports a flag JSON.none_as_null which when set to True will result in the Python constant `None` evaluating to the value of SQL NULL, and when set to False results in the Python constant `None` evaluating to the value of JSON `"null"`. The Python value `None` may be used in conjunction with either JSON.NULL and null() in order to indicate NULL values, but care must be taken as to the value of the JSON.none_as_null in these cases.

Customizing the JSON Serializer

The JSON serializer and deserializer used by JSON defaults to Python's `json.dumps` and `json.loads` functions; in the case of the psycopg2 dialect, psycopg2 may be using its own custom loader function.

In order to affect the serializer / deserializer, they are currently configurable at the create_engine() level via the create_engine.json_serializer and create_engine.json_deserializer parameters. For example, to turn off

```
ensure_ascii:

engine = create_engine(
    "sqlite://",
    json_serializer=lambda obj: json.dumps(obj, ensure_ascii=False))
```

Changed in version 1.3.7: SQLite dialect's `json_serializer` and `json_deserializer` parameters renamed from `_json_serializer` and `_json_deserializer`.

New in version 1.1.

class `Comparator`(*expr*)¶

Bases: `sqlalchemy.types.Comparator`, `sqlalchemy.types.Comparator`

Define comparison operations for `types.JSON`.

`as_boolean`()¶

Cast an indexed value as boolean.

e.g.:

```
stmt = select([
    mytable.c.json_column['some_data'].as_boolean()
]).where(
    mytable.c.json_column['some_data'].as_boolean() == True
)
```

New in version 1.3.11.

`as_float`()¶

Cast an indexed value as float.

e.g.:

```
stmt = select([
    mytable.c.json_column['some_data'].as_float()
]).where(
    mytable.c.json_column['some_data'].as_float() == 29.75
)
```

New in version 1.3.11.

`as_integer`()¶

Cast an indexed value as integer.

e.g.:

```
stmt = select([
    mytable.c.json_column['some_data'].as_integer()
]).where(
    mytable.c.json_column['some_data'].as_integer() == 5
)
```

New in version 1.3.11.

`as_json`()¶

Cast an indexed value as JSON.

This is the default behavior of indexed elements in any case.

Note that comparison of full JSON structures may not be supported by all backends.

New in version 1.3.11.

`as_string`()¶

Cast an indexed value as string.

e.g.:

```
stmt = select([
    mytable.c.json_column['some_data'].as_string()
]).where(
    mytable.c.json_column['some_data'].as_string() ==
    'some string'
)
```

New in version 1.3.11.

class `JSONElementType`¶

Bases: `sqlalchemy.types.TypeEngine`

common function for index / path elements in a JSON expression.

`bind_processor`(*dialect*)¶

Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

Parameters

dialect¶ – Dialect instance in use.

`literal_processor`(*dialect*)¶

Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the “literal_binds” flag, typically used in DDL generation as well as in certain scenarios where backends don’t accept bound parameters.

New in version 0.9.0.

class `JSONIndexType`¶

Bases: `sqlalchemy.types.JSONElementType`

Placeholder for the datatype of a JSON index value.

This allows execution-time processing of JSON index values for special syntaxes.

class `JSONPathType`¶

Bases: `sqlalchemy.types.JSONElementType`

Placeholder type for JSON path operations.

This allows execution-time processing of a path-based index value into a specific SQL syntax.

`NULL` = *symbol('JSON_NULL')*¶

Describe the json value of NULL.

This value is used to force the JSON value of `"null"` to be used as the value. A value of Python `None` will be recognized either as SQL NULL or JSON `"null"`, based on the setting of the `JSON.none_as_null` flag; the `JSON.NULL` constant can be used to always resolve to JSON `"null"` regardless of this setting. This is in contrast to the `sql.null()` construct, which always resolves to SQL NULL. E.g.:

```
from sqlalchemy import null
from sqlalchemy.dialects.postgresql import JSON

# will *always* insert SQL NULL
obj1 = MyObject(json_value=null())

# will *always* insert JSON string "null"
obj2 = MyObject(json_value=JSON.NULL)

session.add_all([obj1, obj2])
session.commit()
```

In order to set JSON NULL as a default value for a column, the most transparent method is to use `text()`:

```
Table(
    'my_table', metadata,
    Column('json_data', JSON, default=text("'null'"))
)
```

While it is possible to use `JSON.NULL` in this context, the `JSON.NULL` value will be returned as the value of the column, which in the context of the ORM or other repurposing of the default value, may not be desirable. Using a SQL expression means the value will be re-fetched from the database within the context of retrieving generated defaults.

`__init__`(*none_as_null=False*)¶

Construct a `types.JSON` type.

Parameters

none_as_null=False¶ –

if True, persist the value `None` as a SQL NULL value, not the JSON encoding of `null`. Note that when this flag is False, the `null()` construct can still be used to persist a NULL value:

```
from sqlalchemy import null
conn.execute(table.insert(), data=null())
```

`bind_processor`(*dialect*)¶

Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

Parameters

dialect¶ – Dialect instance in use.

`comparator_factory`¶

alias of `JSON.Comparator`

`python_type`¶

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`result_processor`(*dialect*, *coltype*)¶

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect¶** – Dialect instance in use.
- **coltype¶** – DBAPI coltype argument received in cursor.description.

`should_evaluate_none¶`

Alias of `JSON.none_as_null`

class `sqlalchemy.types.INTEGER¶`

Bases: `sqlalchemy.types.Integer`

The SQL INT or INTEGER type.

class `sqlalchemy.types.NCHAR`(*length=None*, ***kwargs*)¶

Bases: `sqlalchemy.types.Unicode`

The SQL NCHAR type.

class `sqlalchemy.types.NVARCHAR`(*length=None*, ***kwargs*)¶

Bases: `sqlalchemy.types.Unicode`

The SQL NVARCHAR type.

class `sqlalchemy.types.NUMERIC`(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)¶

Bases: `sqlalchemy.types.Numeric`

The SQL NUMERIC type.

class `sqlalchemy.types.REAL`(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*)¶

Bases: `sqlalchemy.types.Float`

The SQL REAL type.

class `sqlalchemy.types.SMALLINT¶`

Bases: `sqlalchemy.types.SmallInteger`

The SQL SMALLINT type.

class `sqlalchemy.types.TEXT`(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*, *_expect_unicode=False*)¶

Bases: `sqlalchemy.types.Text`

The SQL TEXT type.

class sqlalchemy.types.TIME(*timezone=False*)¶

Bases: sqlalchemy.types.Time

The SQL TIME type.

class sqlalchemy.types.TIMESTAMP(*timezone=False*)¶

Bases: sqlalchemy.types.DateTime

The SQL TIMESTAMP type.

TIMESTAMP datatypes have support for timezone storage on some backends, such as PostgreSQL and Oracle. Use the timezone argument in order to enable "TIMESTAMP WITH TIMEZONE" for these backends.

__init__(*timezone=False*)¶

Construct a new TIMESTAMP.

Parameters

timezone¶ – boolean. Indicates that the TIMESTAMP type should enable timezone support, if available on the target database. On a per-dialect basis is similar to "TIMESTAMP WITH TIMEZONE". If the target database does not support timezones, this flag is ignored.

get_dbapi_type(*dbapi*)¶

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling setinputsizes(), for example.

class sqlalchemy.types.VARBINARY(*length=None*)¶

Bases: sqlalchemy.types._Binary

The SQL VARBINARY type.

class sqlalchemy.types.VARCHAR(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*, *_expect_unicode=False*)¶

Bases: sqlalchemy.types.String

The SQL VARCHAR type.

Vendor-Specific Types¶

Database-specific types are also available for import from each database's dialect module. See the Dialects reference for the database you're interested in.

For example, MySQL has a BIGINT type and PostgreSQL has an INET type. To use these, import them from the module explicitly:

```
from sqlalchemy.dialects import mysql

table = Table('foo', metadata,
    Column('id', mysql.BIGINT),
    Column('enumerates', mysql.ENUM('a', 'b', 'c'))
)
```

Or some PostgreSQL types:

```
from sqlalchemy.dialects import postgresql

table = Table('foo', metadata,
    Column('ipaddress', postgresql.INET),
    Column('elements', postgresql.ARRAY(String))
)
```

Each dialect provides the full set of typenames supported by that backend within its *__all__* collection, so that a simple *import* * or similar will import all supported types as implemented for that backend:

```
from sqlalchemy.dialects.postgresql import *

t = Table('mytable', metadata,
            Column('id', INTEGER, primary_key=True),
            Column('name', VARCHAR(300)),
            Column('inetaddr', INET)
)
```

Where above, the INTEGER and VARCHAR types are ultimately from sqlalchemy.types, and INET is specific to the PostgreSQL dialect.

Some dialect level types have the same name as the SQL standard type, but also provide additional arguments. For example, MySQL implements the full range of character and string types including additional arguments such as *collation* and *charset*:

```
from sqlalchemy.dialects.mysql import VARCHAR, TEXT

table = Table('foo', meta,
    Column('col1', VARCHAR(200, collation='binary')),
    Column('col2', TEXT(charset='latin1'))
)
```