

---

# **Sqeleton Documentation**

**Erez Shinan**

**Apr 03, 2023**



## API REFERENCE

<b>1</b>	<b>Database drivers</b>	<b>3</b>
<b>2</b>	<b>Connection editor</b>	<b>5</b>
<b>3</b>	<b>What's next?</b>	<b>7</b>
<b>4</b>	<b>Introduction to Sqeleton</b>	<b>9</b>
4.1	Database Interface . . . . .	9
4.2	Query Builder . . . . .	11
4.3	Advanced topics . . . . .	16
4.4	Other features . . . . .	19
<b>5</b>	<b>List of supported databases</b>	<b>21</b>
<b>6</b>	<b>Connection Editor</b>	<b>23</b>
6.1	Install . . . . .	23
6.2	Run . . . . .	23
<b>7</b>	<b>Python API Reference</b>	<b>25</b>
7.1	User API . . . . .	25
7.2	Internals . . . . .	30
<b>8</b>	<b>Sqeleton</b>	<b>43</b>
<b>9</b>	<b>Resources</b>	<b>45</b>
	<b>Python Module Index</b>	<b>47</b>
	<b>Index</b>	<b>49</b>



Sqeleton can be installed using pip:

```
pip install sqeleton
```



## DATABASE DRIVERS

To ensure that the database drivers are compatible with `sqeleon`, we recommend installing them along with `sqeleon`, using `pip`'s `[]` syntax:

- `pip install 'sqeleon[mysql]'`
- `pip install 'sqeleon[postgresql]'`
- `pip install 'sqeleon[snowflake]'`
- `pip install 'sqeleon[presto]'`
- `pip install 'sqeleon[oracle]'`
- `pip install 'sqeleon[trino]'`
- `pip install 'sqeleon[clickhouse]'`
- `pip install 'sqeleon[vertica]'`
- For BigQuery, see: <https://pypi.org/project/google-cloud-bigquery/>

*Some drivers have dependencies that cannot be installed using `pip` and still need to be installed manually.*

It is also possible to install several databases at once. For example:

```
pip install 'sqeleon[mysql, postgresql]'
```

Note: Some shells use `"` for escaping instead, like:

```
pip install "sqeleon[mysql, postgresql]"
```





## CONNECTION EDITOR

Skeleton provides a TUI connection editor, that can be installed using:

```
pip install 'sqeleton[tui]'
```

Read more [here](#).



## WHAT'S NEXT?

Read the *introduction* and start coding!



## INTRODUCTION TO SKELETON

Skeleton is a Python library for querying SQL databases.

It has two complementary APIs:

1. *Database interface*, for connecting and querying.
2. *Query builder*, for constructing query expressions (ASTs).

The following sections delve into each one of them.

### 4.1 Database Interface

Skeleton's database interface enables provides a unified interface for accessing a wide array of databases.

Each database has its own dialect, which tells the compiler which SQL to produce for each expression. Extra features can be added to the dialect using *mixins*.

#### 4.1.1 connect()

Database instances are created using the `connect()` function, when given a URI or equivalent dict:

```
>>> from skeleton import connect
>>> connect("mysql://localhost/db")
<skeleton.databases.mysql.MySQL object at ...>
>>> connect({"driver": "mysql", "host": "localhost", "database": "db"})
<skeleton.databases.mysql.MySQL object at ...>
```

For non-cloud DBs, you can also provide the size of the local thread pool, i.e. how many worker-threads it should spawn.

Database instances have a few useful methods, like `list_tables()`, `query_table_schema()`:

```
>>> db = connect("postgresql:///")
>>> db.name
'PostgreSQL'
>>> db.list_tables('R%')
[('Rating',), ('Rating_del1',), ('Rating_del1p',), ('Rating_update001p',), ('Rating_
↪update1',), ('Rating_update1p',), ('Rating_update50p',)]
>>> db.close()
>>> db.is_closed
True
```

### 4.1.2 Database.query()

The query function accepts either raw SQL, or a query expression, created by the query-builder:

```
>>> db = connect("duckdb://:memory:")

# Query raw SQL
>>> db.query('select 1 + 1')
[(2,)]

# Query SQL expression (more on that later)
>>> from sqeleton.queries import current_timestamp
>>> db.query(current_timestamp())
datetime.datetime(2022, 12, 28, 17, 1, 11)
```

It also accepts a list of statements:

```
from sqeleton.queries import table
>>> tmp = table('tmp', schema={'i': int})
# Query list of statements, returns the last result
>>> db.query([
...     tmp.create(),
...     tmp.insert_rows([x] for x in range(100)),
...     'select sum(i) from tmp',
... ])
[(4950,)]
```

It's possible to tell the query() method the type you expect as a result, and it will validate and attempt to cast it:

```
# Same as default. Return a list of tuples.
>>> db.query('select 1 + 1', list)
[(2,)]
# Return a tuple. Expects only 1 row returned.
>>> db.query('select 1 + 1', tuple)
(2,)
# Return an int. Expects only 1 row and column returned.
>>> db.query('select 1 + 1', int)
2
```

## 4.2 Query Builder

Sqeleton's query-builder takes a lot of ideas from both SQLAlchemy and PyPika, however there are several notable differences.

### 4.2.1 table()

To query tables, users create a table instance, and use chain methods to build the query expression:

```
>>> from sqeleton import table

# SELECT * FROM tmp -- Query everything in table 'tmp'
>>> expr = table('tmp').select()
>>> db.query(expr)
[(0,), (1,), (2,), (3,), ...]
```

To specify a dotted table path (for schemas, etc.), use either form:

- `table('foo', 'bar')` compiles to `foo.bar`.
- `table(('foo', 'bar'))` also compiles to `foo.bar`.

### 4.2.2 table attributes and this

In sqeleton, table attributes are accessed either through the table's `[]` operator, or using the `this` object. When compiled, `this.foo` evaluates to the column of the attached table:

```
from sqeleton import table, this

tbl = table('tmp')

# All exprs are equivalent to: SELECT foo, bar FROM tmp
expr1 = tbl.select(tbl['foo'], tbl['bar'])
expr1 = tbl.select(this['foo'], this['bar'])
expr2 = tbl.select(this.foo, this.bar)
```

It's also possible to give `this` a list of attributes:

```
attrs = ['foo', 'bar']

# Equivalent to: SELECT foo, bar FROM tbl
expr = tbl.select(*this[attrs])
```

It's recommended to prefer the `this.attr` syntax whenever possible, because it's shorter, more readable, and more amenable to refactoring.

However, the `[]` syntax is very useful for variable attribute names.

### 4.2.3 .select(), .where(), .order\_by(), .limit()

These are fairly straightforward, and similar to other libraries:

```
# SELECT a FROM tmp WHERE b > 10 ORDER BY c LIMIT 20
table('tmp').select(this.a).where(this.b > 10).order_by(this.c).limit(20)
```

There are a few things worth mentioning:

- **SKIP** - You can provide the `sqeleton.SKIP` keyword to these functions, which does nothing. That can be useful for chaining conditionals. For example:

```
from sqeleton import SKIP

# SELECT name[, age] FROM tmp [WHERE age > 18] [LIMIT limit]
rows = (table('tmp')
        .select(this.name, (this.age if include_age else SKIP))
        .where(this.age > 18 if only_adults else SKIP)
        .limit(limit if limit is not None else SKIP)
        )
```

To alias columns in `.select()`, use keyword arguments:

```
# SELECT id, (first || ' ' || last) AS full_name, (age >= 18) AS is_adult FROM person
table('person').select(
    this.id,
    full_name = this.first + " " + this.last,
    is_adult = self.age >= 18
)
```

- **Generators**

It's possible to provide generators to `select()`, `where()`, and `order_by()`, enabling syntax like this:

```
fields = 'foo', 'bar', '_xyz'

# SELECT foo, bar FROM tmp WHERE foo > 0 AND bar > 0 AND _xyz > 0
all_above_0 = (table('tmp')
               .select(this[f] for f in fields if not f.startswith('_'))
               .where(this[f] > 0 for f in fields)
               )
```

### 4.2.4 .group\_by(), .agg(), and .having()

Group-by in `sqeleton` behaves like in SQL, except for a small change in syntax:

```
# SELECT a, sum(b) FROM tmp GROUP BY 1
table('tmp').group_by(this.a).agg(this.b.sum())

# SELECT a, sum(b) FROM a GROUP BY 1 HAVING (b > 10)
(table('tmp')
 .group_by(this.a)
 .agg(this.b.sum())
```

(continues on next page)



(continued from previous page)

```

        .having(this.b > 10)
    )

```

These functions also accept generators and SKIP.

A call to `.agg()` must follow every call to `group_by()`.

Any use of `.select()` will be considered a separate sub-query:

```

# SELECT (c + 1) FROM (SELECT b, c FROM (SELECT a FROM tmp) GROUP BY 1)
rows = (table('tmp')
        .select(this.a)
        .group_by(this.b)
        .agg(this.c)
        .select(this.c + 1)
    )

```

## 4.2.5 More table operations

Tables and queries also support the following methods:

```

# SELECT count() FROM a`
a.count()

# SELECT * FROM a UNION b
a.union(b)

# SELECT * FROM a UNION ALL b -- concat tables
a.union_all(b)

# SELECT * FROM a EXCEPT b      -- or MINUS
a.minus(b)

# select * from a INTERSECT b
a.intersect(b)

```

## 4.2.6 .join(), .on()

When joining, it's recommended to use explicit tables names, instead of `this`, in order to avoid potential name collisions.

```

person = table('person')
city = table('city')

name_and_city = (
    person
    .join(city)
    .on(person['city_id'] == city['id'])
    .select(person['id'], city['name'])
)

```

`.on()` also supports generators and SKIP.

### 4.2.7 when() and .then()

Sqeleton provides a way to construct case-when-then expressions:

```
from sqeleton.queries import when

# SELECT CASE
#   WHEN (type = 'text') THEN text
#   WHEN (type = 'number') THEN number
#   ELSE 'unknown type' END
# FROM foo
rows = table('foo').select(
    when(this.type == 'text').then(this.text)
    .when(this.type == 'number').then(this.number)
    .else_('unknown type')
)
```

Each `.when()` must be followed by a `.then()`.

Sqeleton also provides a convenience `if_()` function for simple conditionals:

```
from sqeleton import if_

# SELECT CASE WHEN b THEN c ELSE d END FROM foo
table('foo').select(if_(b, c, d))
```

### 4.2.8 DDL - .create(), .drop(), .truncate()

These methods create DDL expressions (or “data-definition language”).

To execute these expressions on a database, you have to call `Database.query()`.

- `.create()`

It’s possible to create empty new tables using a schema, or new tables populated using a query.

Both are done by using the `.create()` method:

```
source_table = table('source')

db.query([

    # CREATE TABLE new AS SELECT * FROM source
    table('new').create(source_table),

    # CREATE TABLE new AS SELECT * FROM source WHERE x > 0
    table('new_nonzero').create(source_table.where(this.x > 0)),

    # CREATE TABLE foo (id INT, name VARCHAR)
    table('foo', schema={
        id: int,
        name: str
    }).create()

])
```

The `.create()` method also accepts the following keyword parameters:

- `if_not_exists` - Adds `IF NOT EXISTS` to the create statement
- `primary_keys` - Specify primary keys when creating the table
- `.drop()`, `.truncate()`

These are the simple parallels of `DROP TABLE` and `TRUNCATE TABLE`.

#### 4.2.9 Add data - `.insert_row()`, `.insert_rows()`, `.insert_expr()`

These methods insert rows of constant values (from Python), or from a query expression.

```
# INSERT INTO atoms VALUES ('H', 1)
table('atoms').insert_row("H", 1)

# INSERT INTO atoms VALUES ('H', 1), ('He', 2)
rows = [
    ("H", 1),
    ("He", 2)
]
table('atoms').insert_rows(rows)

# INSERT INTO foo SELECT * FROM bar      -- Concat 'bar' to 'foo'
table('foo').insert_expr(table('bar'))
```

A common pattern is to call `.insert_rows()` with a generator:

```
rows = {
    1: "H",
    2: "He",
}
table('atoms').insert_rows((sym, num) for num, sym in rows.items())
```

#### 4.2.10 Raw SQL using `code()`

It's possible to combine Sqeleton's query expressions with raw SQL code.

It allows users to use features and syntax that Sqeleton doesn't yet support.

Keep in mind that the code is very unlikely to port to other databases, so if you need to support more than one database, keep your use of `code()` to a minimum, and use it behind abstracting functions.

```
from sqeleton import code

# SELECT b, <x> FROM tmp WHERE <y>
table('tmp').select(this.b, code("<x>")).where(code("<y>"))
```

It's the user's responsibility to make sure the contents of the string given to `code()` are correct and safe for execution.

Strings given to `code()` are actually templates, and can embed query expressions given as arguments:

```
def tablesample(tbl, size):
    return code("{tbl} TABLESAMPLE BERNOULLI ({size})", tbl=tbl, size=size)
```

(continues on next page)

(continued from previous page)

```
nonzero = table('points').where(this.x > 0, this.y > 0)

# SELECT * FROM points WHERE (x > 0) AND (y > 0) TABLESAMPLE BERNOULLI (10)
sample_expr = tablesample(nonzero)
```

## 4.2.11 Bound Tables & Expressions - Database.table()

Sqeleton's query expressions are database-agnostic, which makes it easy to run the exact same queries on different databases.

While this functional style is sufficient for most purposes, it's sometimes convenient to have a more object-oriented approach, and pass around query expressions bound to a specific database. That can be especially useful when running all the queries on the same database, or when different databases need drastically different queries.

Bound exprs support `.query()`, which will execute them on the bound database:

```
# SELECT foo FROM tmp
db.table('tmp').select(this.foo).query()

# SELECT foo FROM tmp -- expects one row and one column
db.table('tmp').select(this.foo).query(int)
```

Having bound tables, specifically, allow to add the useful `.query_schema()` API:

```
# CREATE TABLE a (b FLOAT)
>>> schema = {'b': float}
>>> db.table('a', schema=schema).create().query()

# Queries the database for the schema, and returns a new bound table instance
>>> t2 = db.table('a').query_schema()
>>> t2.schema
{'b': Float(precision=5)}
```

## 4.3 Advanced topics

### 4.3.1 Dialect Mixins

In Sqeleton, each dialect class represents a specific SQL dialect. Dialects are responsible for providing code fragments to the SQL compiler.

Since Sqeleton aims to support a growing amount of features, and allow custom database implementations, extra features are provided through mixins. That way, when implementing a new database (either in Sqeleton, or in a private code-base), we can pick and choose which features we want to implement, and which ones we don't. Sqeleton will throw an error if the mixin we're trying to use isn't supported by one of the databases we're using.

The simplest way to load mixins is to use the `Connect.load_mixins()` methods, and provide the abstract mixins you want to use:

```
import sqeleton
from sqeleton.abcs.mixins import AbstractMixin_NormalizeValue, AbstractMixin_RandomSample

connect = sqeleton.connect.load_mixins(AbstractMixin_NormalizeValue)
ddb = connect("duckdb://:memory:")
print(ddb.dialect.normalize_boolean("bool", None))
# Outputs:
#   bool::INTEGER::VARCHAR
```

Each database is already aware of the available mixin implementations, because it was defined with the MIXINS attribute. We're only using the abstract mixins to select the mixins we want to use.

## List of mixins

List of available abstract mixins:

- AbstractMixin\_NormalizeValue
- AbstractMixin\_MD5
- AbstractMixin\_Schema
- AbstractMixin\_Regex
- AbstractMixin\_RandomSample
- AbstractMixin\_TimeTravel - Only snowflake & bigquery
- AbstractMixin\_OptimizerHints - Only oracle & mysql

More will be added in the future.

Note that it's still possible to use user-defined mixins that aren't on this list.

## Unimplemented Mixins

Trying to load a mixin that isn't implemented by all databases, will fail:

```
>>> from sqeleton.abcs.mixins import AbstractMixin_TimeTravel
>>> connect.load_mixins(AbstractMixin_TimeTravel)
Traceback (most recent call last):
...
TypeError: Can't instantiate abstract class PostgresqlDialect with abstract method time_
↪travel
```

In such a case, it's possible to use `Connect.for_databases()` to only load for a subset of the available databases:

```
# No problem, time travel is implemented in both
# Trying to connect to other databases will fail
connect = sqeleton.connect.for_databases('bigquery', 'snowflake').load_
↪mixins(AbstractMixin_TimeTravel)
```

The `.load_mixins()` method is just a convenience method. It's possible to achieve the same functionality, and with more fine-grained control, using explicit inheritance, and finally creating a new `Connect` object.

Note that both `.load_mixins()` and `.for_databases()` create new instances of `Connect`, and it's okay to have more than one at the same time.

### Type Inference, mypy, etc.

Python's typing module doesn't yet support intersection / multiple-inheritance, and so `.load_mixins()` can't provide the necessary information for type-checking.

The recommended solution is to override the type of the database returned from `connect()`:

```
from sqeleton.abcs import AbstractDialect, AbstractDatabase

class NewAbstractDialect(AbstractDialect, AbstractMixin_NormalizeValue, AbstractMixin_
↳ RandomSample):
    pass

connect = sqeleton.connect.load_mixins(AbstractMixin_NormalizeValue, AbstractMixin_
↳ RandomSample)
ddb: AbstractDatabase[NewAbstractDialect] = connect("duckdb://:memory:")
# ddb.dialect is now known to implement NewAbstractDialect.
```

### 4.3.2 Query interpreter

In addition to query expressions, `Database.query()` can accept a generator, which will behave as an “interpreter”.

The generator executes queries by yielding them.

Using a query interpreter also guarantees that subsequent calls to `.query()` will run in the same session. That can be useful for using temporary tables, or session variables.

Example:

```
def sample_using_temp_table(db: Database, source_table: ITable, sample_size: int):
    "This function creates a temporary table from a query and then samples rows from it"

    results = []

    def _sample_using_temp_table():
        nonlocal results

        yield code("CREATE TEMPORARY TABLE tmp1 AS {source_table}", source_table=source_
↳ table)

        tbl = table('tmp1')
        try:
            results += yield sample(tbl, sample_size)
        finally:
            yield tbl.drop()

    db.query(_sample_using_temp_table())
    return results
```

### 4.3.3 Query params

TODO

## 4.4 Other features

### 4.4.1 SQL client

Sqeleton comes with a simple built-in SQL client, in the form of a REPL, which accepts SQL commands, and a few special commands.

It accepts any database URL that is supported by Sqeleton. That can be useful for querying databases that don't have established clients.

You can call it using `sqeleton repl <url>`.

Example:

```
# Start a REPL session
$ sqeleton repl duckdb:///pii_test.ddb

# Run SQL
DuckDB> select (22::float / 7) as almost_pi

almost_pi
| 3.142857074737549 |
+-----+
1 rows

# Display help
DuckDB> ?

Commands:
?mytable - shows schema of table 'mytable'
* - shows list of all tables
*pattern - shows list of all tables with name like pattern
Otherwise, runs regular SQL query
```





## LIST OF SUPPORTED DATABASES

- : Implemented and thoroughly tested.
- : Implemented, but not thoroughly tested yet.
- : Implementation in progress.
- : Implementation planned. Contributions welcome.

Is your database not listed here? We accept pull-requests!



## CONNECTION EDITOR

A common complaint among new users was the difficulty in setting up the connections.

Connection URLs are admittedly confusing, and editing `.toml` files isn't always straight-forward either.

To ease this initial difficulty, we added a `textual`-based TUI tool to `skeleton`, that allows users to edit configuration files and test the connections while editing them.

### 6.1 Install

This tool needs `textual` to run. You can install it using:

```
pip install 'skeleton[tui]'
```

Make sure you also have drivers for whatever database connection you're going to edit!

### 6.2 Run

Once everything is installed, you can run the editor with the following command:

```
skeleton conn-editor <conf_path.toml>
```

Example:

```
skeleton conn-editor ~/dbs.toml
```

The available actions and hotkeys will be listed in the status bar.

Note: using the connection editor will delete comments and reformat the file!

We recommend backing up the configuration file before editing it.



## PYTHON API REFERENCE

### 7.1 User API

#### 7.1.1 Database

**exception** `skeleton.databases.base.ConnectError`

**exception** `skeleton.databases.base.QueryError`

**class** `skeleton.databases.base.ThreadLocalInterpreter`(*compiler*: [Compiler](#), *gen*: *Generator*)

An interpreter used to execute a sequence of queries within the same thread and cursor.

Useful for cursor-sensitive operations, such as creating a temporary table.

**class** `skeleton.databases.base.Mixin_Schema`

**table\_information**() → *Compilable*

Query to return a table of schema information about existing tables

**list\_tables**(*table\_schema*: *str*, *like*: *Optional[Compilable]* = *None*) → *Compilable*

Query to select the list of tables in the schema. (query return type: `table[str]`)

If 'like' is specified, the value is applied to the table name, using the 'like' operator.

**class** `skeleton.databases.base.Mixin_RandomSample`

**random\_sample\_n**(*tbl*: [AbstractTable](#), *size*: *int*) → *AbstractTable*

Take a random sample of the given size, i.e. return 'size' amount of rows

**random\_sample\_ratio\_approx**(*tbl*: [AbstractTable](#), *ratio*: *float*) → *AbstractTable*

Take a random sample of the approximate size determined by the ratio (0..1), where 0 means no rows, and 1 means all rows

i.e. the actual amount of rows returned may vary by standard deviation.

**class** `skeleton.databases.base.Mixin_OptimizerHints`

**optimizer\_hints**(*hints*: *str*) → *str*

Creates a compatible `optimizer_hints` string

**Parameters**

**hints** (*optimizer\_hints* - *string of optimizer*) -

**class** `skeleton.databases.base.BaseDialect`

**offset\_limit**(*offset: Optional[int] = None, limit: Optional[int] = None*)

Provide SQL fragment for limit and offset inside a select

**concat**(*items: List[str]*) → str

Provide SQL for concatenating a bunch of columns into a string

**is\_distinct\_from**(*a: str, b: str*) → str

Provide SQL for a comparison where NULL = NULL is true

**timestamp\_value**(*t: datetime*) → str

Provide SQL for the given timestamp value

**random**() → str

Provide SQL for generating a random number between 0..1

**current\_timestamp**() → str

Provide SQL for returning the current timestamp, aka now

**explain\_as\_text**(*query: str*) → str

Provide SQL for explaining a query, returned as table(varchar)

**classmethod load\_mixins**(\**abstract\_mixins*) → Any

Load a list of mixins that implement the given abstract mixins

**class** skeleton.databases.base.**QueryResult**(*rows: list[Any] = <object object at 0x7fb55b8a91a0>, columns: (list[Any]+NoneType) = None*)

**class** skeleton.databases.base.**Database**(\**args*, \*\**kws*)

Base abstract class for databases.

Used for providing connection code and implementation specific SQL utilities.

Instantiated using `connect()`

**query**(*sql\_ast: Union[ExprNode, str, bool, int, float, datetime, ArithString, None, Generator], res\_type: Optional[type] = None*)

Query the given SQL code/AST, and attempt to convert the result to type 'res\_type'

If given a generator, it will execute all the yielded sql queries with the same thread and cursor. The results of the queries are returned by the `yield` stmt (using the `.send()` mechanism). It's a cleaner approach than exposing cursors, but may not be enough in all cases.

**select\_table\_schema**(*path: Tuple[str, ...]*) → str

Provide SQL for selecting the table schema as (name, type, date\_prec, num\_prec)

**query\_table\_schema**(*path: Tuple[str, ...]*) → Dict[str, tuple]

Query the table for its schema for table in 'path', and return {column: tuple} where the tuple is (table\_name, col\_name, type\_repr, datetime\_precision?, numeric\_precision?, numeric\_scale?)

**Note:** This method exists instead of `select_table_schema()`, just because not all databases support accessing the schema using a SQL query.

**select\_table\_unique\_columns**(*path: Tuple[str, ...]*) → str

Provide SQL for selecting the names of unique columns in the table

**query\_table\_unique\_columns**(*path: Tuple[str, ...]*) → List[str]

Query the table for its unique columns for table in 'path', and return {column}

**parse\_table\_name**(*name: str*) → Tuple[str, ...]

Parse the given table name into a DbPath

**close**()

Close connection(s) to the database instance. Querying will stop functioning.

**classmethod load\_mixins**(\**abstract\_mixins*) → type

Extend the dialect with a list of mixins that implement the given abstract mixins.

**class** sqeleton.databases.base.**ThreadedDatabase**(*thread\_count=1*)

Access the database through singleton threads.

Used for database connectors that do not support sharing their connection between different threads.

**abstract create\_connection**()

Return a connection instance, that supports the .cursor() method.

**close**()

Close connection(s) to the database instance. Querying will stop functioning.

**property is\_autocommit: bool**

Return whether the database autocommits changes. When false, COMMIT statements are skipped.

## 7.1.2 Queries

**sqeleton.queries.api.join**(\**tables: ITable*) → *Join*

Inner-join a sequence of table expressions”

When joining, it’s recommended to use explicit tables names, instead of *this*, in order to avoid potential name collisions.

### Example

```
person = table('person')
city = table('city')

name_and_city = (
    join(person, city)
    .on(person['city_id'] == city['id'])
    .select(person['id'], city['name'])
)
```

**sqeleton.queries.api.leftjoin**(\**tables: ITable*)

Left-joins a sequence of table expressions.

See Also: join()

**sqeleton.queries.api.rightjoin**(\**tables: ITable*)

Right-joins a sequence of table expressions.

See Also: join()

**sqeleton.queries.api.outterjoin**(\**tables: ITable*)

Outer-joins a sequence of table expressions.

See Also: join()

`sqeleton.queries.api.cte(expr: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]], *, name: Optional[str] = None, params: Optional[Sequence[str]] = None)`

Define a CTE

`sqeleton.queries.api.table(*path: str, schema: Optional[Union[dict, CaseAwareMapping]] = None) → TablePath`

Defines a table with a path (dotted name), and optionally a schema.

### Parameters

- **path** – A list of names that make up the path to the table.
- **schema** – a dictionary of {name: type}

`sqeleton.queries.api.or_(*exprs: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Apply OR between a sequence of boolean expressions

`sqeleton.queries.api.and_(*exprs: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Apply AND between a sequence of boolean expressions

`sqeleton.queries.api.sum_(expr: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Call SUM(expr)

`sqeleton.queries.api.avg(expr: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Call AVG(expr)

`sqeleton.queries.api.min_(expr: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Call MIN(expr)

`sqeleton.queries.api.max_(expr: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Call MAX(expr)

`sqeleton.queries.api.exists(expr: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Call EXISTS(expr)

`sqeleton.queries.api.if_(cond: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]], then: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]], else_: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]] = None)`

Conditional expression, shortcut to when-then-else.

### Example

```
# SELECT CASE WHEN b THEN c ELSE d END FROM foo
table('foo').select(if_(b, c, d))
```

`sqeleton.queries.api.when(*when_exprs: Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]])`

Start a when-then expression



## Example

```
# SELECT CASE
#   WHEN (type = 'text') THEN text
#   WHEN (type = 'number') THEN number
#   ELSE 'unknown type' END
# FROM foo
rows = table('foo').select(
    when(this.type == 'text').then(this.text)
    .when(this.type == 'number').then(this.number)
    .else_('unknown type')
)
```

`sqeleton.queries.api.coalesce(*exprs)`

Returns a call to COALESCE

`sqeleton.queries.api.current_timestamp()`

Returns CURRENT\_TIMESTAMP() or NOW()

`sqeleton.queries.api.code(code: str, **kw: Dict[str, Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]]]) → Code`

Inline raw SQL code.

It allows users to use features and syntax that Sqeleton doesn't yet support.

It's the user's responsibility to make sure the contents of the string given to `code()` are correct and safe for execution.

Strings given to `code()` are actually templates, and can embed query expressions given as arguments:

### Parameters

- **code** – template string of SQL code. Templated variables are signified with '{var}'.
- **kw** – optional parameters for SQL template.

## Examples

```
# SELECT b, <x> FROM tmp WHERE <y>
table('tmp').select(this.b, code("<x>")).where(code("<y>"))
```

```
def tablesample(tbl, size):
    return code("SELECT * FROM {tbl} TABLESAMPLE BERNOULLI ({size})", tbl=tbl,
    ↪size=size)

nonzero = table('points').where(this.x > 0, this.y > 0)

# SELECT * FROM points WHERE (x > 0) AND (y > 0) TABLESAMPLE BERNOULLI (10)
sample_expr = tablesample(nonzero)
```

## 7.2 Internals

This section is for developers who wish to improve sqeleton, or to extend it within their own project.

Regular users might also find it useful for debugging and understanding, especially at this early stage of the project.

### 7.2.1 Query ASTs

**exception** sqeleton.queries.ast\_classes.**QueryBuilderError**

**exception** sqeleton.queries.ast\_classes.**QB\_TypeError**

**class** sqeleton.queries.ast\_classes.**ExprNode**

Base class for query expression nodes

**class** sqeleton.queries.ast\_classes.**Code**(code: str = <object object at 0x7fb55b8a91a0>, args: (None-Type+dict[(str\*(int+bool+float+ExprNode+NoneType+datetime+ArithString+str)= None)

**class** sqeleton.queries.ast\_classes.**Alias**(expr: (int+bool+float+ExprNode+NoneType+datetime+ArithString+str) = <object object at 0x7fb55b8a91a0>, name: str = <object object at 0x7fb55b8a91a0>)

**class** sqeleton.queries.ast\_classes.**ITable**

**select**(\*exprs, distinct=SKIP, optimizer\_hints=SKIP, \*\*named\_exprs) → *ITable*

Create a new table with the specified fields

**where**(\*exprs)

Filter the rows, based on the given predicates. (aka Selection)

**order\_by**(\*exprs)

Order the rows lexicographically, according to the given expressions.

**limit**(limit: int)

Stop yielding rows after the given limit. i.e. take the first 'n=limit' rows

**join**(target: *ITable*)

Join this table with the target table.

**group\_by**(\*keys) → *GroupBy*

Group according to the given keys.

Must be followed by a call to :ref:GroupBy.agg()

**count**()

SELECT count() FROM self

**union**(other: *ITable*)

SELECT \* FROM self UNION other

**union\_all**(other: *ITable*)

SELECT \* FROM self UNION ALL other

**minus**(other: *ITable*)

SELECT \* FROM self EXCEPT other

**intersect**(*other*: [ITable](#))

SELECT \* FROM self INTERSECT other

**class** `sqeleton.queries.ast_classes.Concat`(*exprs*: *list*[*Any*] = <object object at 0x7fb55b8a91a0>, *sep*: (*NoneType*+*str*) = *None*)

**class** `sqeleton.queries.ast_classes.Count`(*expr*: *NoneType* + *int* + *bool* + *float* + *ExprNode* + *NoneType* + *datetime* + *ArithString* + *str* = *None*, *distinct*: *bool* = *False*)

**type**

alias of `int`

**class** `sqeleton.queries.ast_classes.TestRegex`(*string*: (*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*) = <object object at 0x7fb55b8a91a0>, *pattern*: (*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*) = <object object at 0x7fb55b8a91a0>)

**class** `sqeleton.queries.ast_classes.Func`(*name*: *str* = <object object at 0x7fb55b8a91a0>, *args*: *Sequence*[(*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*)] = <object object at 0x7fb55b8a91a0>)

**class** `sqeleton.queries.ast_classes.WhenThen`(*when*: (*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*) = <object object at 0x7fb55b8a91a0>, *then*: (*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*) = <object object at 0x7fb55b8a91a0>)

**class** `sqeleton.queries.ast_classes.CaseWhen`(*cases*: *Sequence*[*WhenThen*] = <object object at 0x7fb55b8a91a0>, *else\_expr*: (*NoneType*+(*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*)) = *None*)

**when**(\**whens*: *Optional*[*Union*[[ExprNode](#), *str*, *bool*, *int*, *float*, *datetime*, *ArithString*]]) → [QB\\_When](#)

Add a new ‘when’ clause to the case expression

Must be followed by a call to `.then()`

**else\_**(*then*: *Optional*[*Union*[[ExprNode](#), *str*, *bool*, *int*, *float*, *datetime*, *ArithString*]])

Add an ‘else’ clause to the case expression.

Can only be called once!

**class** `sqeleton.queries.ast_classes.QB_When`(*casewhen*: *CaseWhen* = <object object>, *when*: (*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*) = <object object>)

Partial case-when, used for query-building

**then**(*then*: *Optional*[*Union*[[ExprNode](#), *str*, *bool*, *int*, *float*, *datetime*, *ArithString*]]) → [CaseWhen](#)

Add a ‘then’ clause after a ‘when’ was added.

**class** `sqeleton.queries.ast_classes.IsDistinctFrom`(*a*: (*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*) = <object object at 0x7fb55b8a91a0>, *b*: (*int*+*bool*+*float*+*ExprNode*+*NoneType*+*datetime*+*ArithString*+*str*) = <object object at 0x7fb55b8a91a0>)

**type**

alias of bool

```
class skeleton.queries.ast_classes.BinOp(op: str = <object object at 0x7fb55b8a91a0>, args: Sequence[(int+bool+float+ExprNode+NoneType+datetime+ArithString+str)] = <object object at 0x7fb55b8a91a0>)
```

```
class skeleton.queries.ast_classes.UnaryOp(op: str = <object object at 0x7fb55b8a91a0>, expr: (int+bool+float+ExprNode+NoneType+datetime+ArithString+str) = <object object at 0x7fb55b8a91a0>)
```

```
class skeleton.queries.ast_classes.BinBoolOp(op: str = <object object>, args: Sequence[(int+bool+float+ExprNode+NoneType+datetime+ArithString+str)] = <object object>)
```

**type**

alias of bool

```
class skeleton.queries.ast_classes.Column(source_table: ITable = <object object at 0x7fb55b8a91a0>, name: str = <object object at 0x7fb55b8a91a0>)
```

```
class skeleton.queries.ast_classes.TablePath(path: tuple[str] = <object object at 0x7fb55b8a91a0>, schema: (NoneType+CaseAwareMapping) = None)
```

```
create(source_table: Optional[ITable] = None, *, if_not_exists: bool = False, primary_keys: Optional[List[str]] = None)
```

Returns a query expression to create a new table.

### Parameters

- **source\_table** – a table expression to use for initializing the table. If not provided, the table must have a schema specified.
- **if\_not\_exists** – Add a ‘if not exists’ clause or not. (note: not all dbs support it!)
- **primary\_keys** – List of column names which define the primary key

```
drop(if_exists=False)
```

Returns a query expression to delete the table.

### Parameters

- **if\_not\_exists** – Add a ‘if not exists’ clause or not. (note: not all dbs support it!)

```
truncate()
```

Returns a query expression to truncate the table. (remove all rows)

```
insert_rows(rows: Sequence, *, columns: Optional[List[str]] = None)
```

Returns a query expression to insert rows to the table, given as Python values.

### Parameters

- **rows** – A list of tuples. Must all have the same width.
- **columns** – Names of columns being populated. If specified, must have the same length as the tuples.

```
insert_row(*values, columns: Optional[List[str]] = None)
```

Returns a query expression to insert a single row to the table, given as Python values.

### Parameters

**columns** – Names of columns being populated. If specified, must have the same length as ‘values’

**insert\_expr**(*expr*: *Optional[Union[ExprNode, str, bool, int, float, datetime, ArithString]]*)

Returns a query expression to insert rows to the table, given as a query expression.

### Parameters

**expr** – query expression to from which to read the rows

**time\_travel**(*\**, *before*: *bool* = *False*, *timestamp*: *Optional[datetime]* = *None*, *offset*: *Optional[int]* = *None*, *statement*: *Optional[str]* = *None*) → *Compilable*

Selects historical data from the table

### Parameters

- **before** – If false, inclusive of the specified point in time. If True, only return the time before it. (at/before)
- **timestamp** – A constant timestamp
- **offset** – the time ‘offset’ seconds before now
- **statement** – identifier for statement, e.g. query ID

Must specify exactly one of *timestamp*, *offset* or *statement*.

```
class skeleton.queries.ast_classes.TableAlias(source_table: ITable = <object object at
0x7fb55b8a91a0>, name: str = <object object at
0x7fb55b8a91a0>)
```

```
class skeleton.queries.ast_classes.Join(source_tables: Sequence[ITable] = <object object at
0x7fb55b8a91a0>, op: (NoneType+str) = None, on_exprs:
(None-
Type+Sequence[(int+bool+float+ExprNode+NoneType+datetime+ArithString+str
= None, columns: (None-
Type+Sequence[(int+bool+float+ExprNode+NoneType+datetime+ArithString+str
= None)
```

**on**(*\*exprs*) → *Join*

Add an ON clause, for filtering the result of the cartesian product (i.e. the JOIN)

**select**(*\*exprs*, *\*\*named\_exprs*) → *ITable*

Select fields to return from the JOIN operation

See Also: `ITable.select()`

```
class skeleton.queries.ast_classes.GroupBy(table: ITable = <object object at 0x7fb55b8a91a0>, keys:
(None-
Type+Sequence[(int+bool+float+ExprNode+NoneType+datetime+ArithString
= None, values: (None-
Type+Sequence[(int+bool+float+ExprNode+NoneType+datetime+ArithString
= None, having_exprs: (None-
Type+Sequence[(int+bool+float+ExprNode+NoneType+datetime+ArithString
= None)
```

**having**(*\*exprs*)

Add a ‘HAVING’ clause to the group-by

**agg**(\*exprs)

Select aggregated fields for the group-by.

```
class skeleton.queries.ast_classes.TableOp(op: str = <object object at 0x7fb55b8a91a0>, table1:
    ITable = <object object at 0x7fb55b8a91a0>, table2: ITable
    = <object object at 0x7fb55b8a91a0>)
```

```
class skeleton.queries.ast_classes.Select(table: NoneType + int + bool + float + ExprNode +
    NoneType + datetime + ArithString + str = None, columns:
    NoneType + Sequence[int + bool + float + ExprNode +
    NoneType + datetime + ArithString + str] = None,
    where_exprs: NoneType + Sequence[int + bool + float +
    ExprNode + NoneType + datetime + ArithString + str] =
    None, order_by_exprs: NoneType + Sequence[int + bool +
    float + ExprNode + NoneType + datetime + ArithString + str]
    = None, group_by_exprs: NoneType + Sequence[int + bool +
    float + ExprNode + NoneType + datetime + ArithString + str]
    = None, having_exprs: NoneType + Sequence[int + bool +
    float + ExprNode + NoneType + datetime + ArithString + str]
    = None, limit_expr: int + NoneType = None, distinct: bool =
    False, optimizer_hints: NoneType + Sequence[int + bool +
    float + ExprNode + NoneType + datetime + ArithString + str]
    = None)
```

```
class skeleton.queries.ast_classes.Cte(source_table:
    (int+bool+float+ExprNode+NoneType+datetime+ArithString+str)
    = <object object at 0x7fb55b8a91a0>, name: (NoneType+str) =
    None, params: (Sequence[str]+NoneType) = None)
```

```
class skeleton.queries.ast_classes.This
```

Builder object for accessing table attributes.

Automatically evaluates to the the ‘top-most’ table during compilation.

```
class skeleton.queries.ast_classes.In(expr:
    (int+bool+float+ExprNode+NoneType+datetime+ArithString+str)
    = <object object at 0x7fb55b8a91a0>, list: Se-
    quence[(int+bool+float+ExprNode+NoneType+datetime+ArithString+str)]
    = <object object at 0x7fb55b8a91a0>)
```

**type**

alias of bool

```
class skeleton.queries.ast_classes.Cast(expr:
    (int+bool+float+ExprNode+NoneType+datetime+ArithString+str)
    = <object object at 0x7fb55b8a91a0>, target_type:
    (int+bool+float+ExprNode+NoneType+datetime+ArithString+str)
    = <object object at 0x7fb55b8a91a0>)
```

```
class skeleton.queries.ast_classes.Random
```

**type**

alias of float

```
class skeleton.queries.ast_classes.ConstantTable(rows: Sequence[Sequence[Any]] = <object object at
    0x7fb55b8a91a0>)
```

```
class sqeleton.queries.ast_classes.Explain(select: Select = <object object at 0x7fb55b8a91a0>)
```

```
    type
```

```
        alias of str
```

```
class sqeleton.queries.ast_classes.CurrentTimestamp
```

```
    type
```

```
        alias of datetime
```

```
class sqeleton.queries.ast_classes.TimeTravel(table: TablePath = <object object at 0x7fb55b8a91a0>,
                                              before: bool = False, timestamp: (NoneType+datetime)
                                              = None, offset: (int+NoneType) = None, statement:
                                              (NoneType+str) = None)
```

```
class sqeleton.queries.ast_classes.Statement
```

```
class sqeleton.queries.ast_classes.CreateTable(path: TablePath = <object object at 0x7fb55b8a91a0>,
                                              source_table: (None-
                                              Type+(int+bool+float+ExprNode+NoneType+datetime+ArithString+str)
                                              = None, if_not_exists: bool = False, primary_keys:
                                              (NoneType+list[str]) = None)
```

```
class sqeleton.queries.ast_classes.DropTable(path: TablePath = <object object at 0x7fb55b8a91a0>,
                                              if_exists: bool = False)
```

```
class sqeleton.queries.ast_classes.TruncateTable(path: TablePath = <object object at
0x7fb55b8a91a0>)
```

```
class sqeleton.queries.ast_classes.InsertToTable(path: TablePath = <object object at
0x7fb55b8a91a0>, expr:
(int+bool+float+ExprNode+NoneType+datetime+ArithString+str)
= <object object at 0x7fb55b8a91a0>, columns:
(NoneType+list[str]) = None, returning_exprs:
(NoneType+list[str]) = None)
```

```
    returning(*exprs)
```

```
        Add a 'RETURNING' clause to the insert expression.
```

```
        Note: Not all databases support this feature!
```

```
class sqeleton.queries.ast_classes.Commit
```

```
    Generate a COMMIT statement, if we're in the middle of a transaction, or in auto-commit. Otherwise SKIP.
```

```
class sqeleton.queries.ast_classes.Param(name: str = <object object>)
```

```
    A value placeholder, to be specified at compilation time using the cv_params context variable.
```

## 7.2.2 Query Compiler

```
exception sqeleton.queries.compiler.CompileError
```

```
class sqeleton.queries.compiler.Root
```

```
    Nodes inheriting from Root can be used as root statements in SQL (e.g. SELECT yes, RANDOM() no)
```

```
class sqeleton.queries.compiler.Compiler(database: AbstractDatabase = <object object at
0x7fb55b8a91a0>, params: dict[(Any*Any)] = <factory>,
in_select: bool = False, in_join: bool = False, _table_context:
list[Any] = <factory>, _subqueries: dict[(str*Any)] =
<factory>, root: bool = True, _counter: list[Any] =
<factory>)
```

### 7.2.3 ABCS

```
class sqeleton.abcs.database_types.PrecisionType(precision: int = <object object at 0x7fb55b8a91a0>,
rounds: (bool+Unknown) = Unknown)
```

```
class sqeleton.abcs.database_types.TemporalType(precision: int = <object object>, rounds:
(bool+Unknown) = Unknown)
```

```
class sqeleton.abcs.database_types.Timestamp(precision: int = <object object>, rounds:
(bool+Unknown) = Unknown)
```

```
class sqeleton.abcs.database_types.TimestampTZ(precision: int = <object object>, rounds:
(bool+Unknown) = Unknown)
```

```
class sqeleton.abcs.database_types.Datetime(precision: int = <object object>, rounds: (bool+Unknown)
= Unknown)
```

```
class sqeleton.abcs.database_types.Date(precision: int = <object object>, rounds: (bool+Unknown) =
Unknown)
```

```
class sqeleton.abcs.database_types.NumericType(precision: int = <object object at 0x7fb55b8a91a0>)
```

```
class sqeleton.abcs.database_types.FractionalType(precision: int = <object object>)
```

```
class sqeleton.abcs.database_types.Float(precision: int = <object object>)
```

**python\_type**

alias of float

```
class sqeleton.abcs.database_types.IKey
```

Interface for ColType, for using a column as a key in table.

**abstract property python\_type: type**

Return the equivalent Python type of the key

```
class sqeleton.abcs.database_types.Decimal(precision: int = <object object>)
```

**property python\_type: type**

Return the equivalent Python type of the key

```
class sqeleton.abcs.database_types.StringType
```

**python\_type**

alias of str

```
class sqeleton.abcs.database_types.ColType_UUID
```

**python\_type**

alias of ArithUUID



```

class sqeleton.abcs.database_types.ColType_Alphanum
    python_type
        alias of ArithAlphanumeric
class sqeleton.abcs.database_types.Native_UUID
class sqeleton.abcs.database_types.String_UUID
class sqeleton.abcs.database_types.String_Alphanum
class sqeleton.abcs.database_types.String_VaryingAlphanum
class sqeleton.abcs.database_types.String_FixedAlphanum(length: int = <object object at 0x7fb55b8a91a0>)

class sqeleton.abcs.database_types.Text
class sqeleton.abcs.database_types.Integer(precision: int = 0, python_type: type = <class 'int'>)
    python_type
        alias of int
class sqeleton.abcs.database_types.UnknownColType(text: str = <object object at 0x7fb55b8a91a0>)
class sqeleton.abcs.database_types.AbstractDialect
    Dialect-dependent query expressions
    abstract property name: str
        Name of the dialect
    abstract classmethod load_mixins(*abstract_mixins) → Any
        Load a list of mixins that implement the given abstract mixins
    abstract property ROUNDS_ON_PREC_LOSS: bool
        True if db rounds real values when losing precision, False if it truncates.
    abstract quote(s: str)
        Quote SQL name
    abstract concat(items: List[str]) → str
        Provide SQL for concatenating a bunch of columns into a string
    abstract is_distinct_from(a: str, b: str) → str
        Provide SQL for a comparison where NULL = NULL is true
    abstract to_string(s: str) → str
        Provide SQL for casting a column to string
    abstract random() → str
        Provide SQL for generating a random number between 0..1
    abstract current_timestamp() → str
        Provide SQL for returning the current timestamp, aka now
    abstract offset_limit(offset: Optional[int] = None, limit: Optional[int] = None)
        Provide SQL fragment for limit and offset inside a select

```

**abstract explain\_as\_text**(*query: str*) → str

Provide SQL for explaining a query, returned as table(varchar)

**abstract timestamp\_value**(*t: datetime*) → str

Provide SQL for the given timestamp value

**abstract set\_timezone\_to\_utc**() → str

Provide SQL for setting the session timezone to UTC

**abstract parse\_type**(*table\_path: Tuple[str, ...], col\_name: str, type\_repr: str, datetime\_precision: Optional[int] = None, numeric\_precision: Optional[int] = None, numeric\_scale: Optional[int] = None*) → ColType

Parse type info as returned by the database

**class** skeleton.abcs.database\_types.**AbstractDatabase**(\*args, \*\*kwargs)

**abstract property dialect: T\_Dialect**

The dialect of the database. Used internally by Database, and also available publicly.

**abstract classmethod load\_mixins**(\*abstract\_mixins) → type

Extend the dialect with a list of mixins that implement the given abstract mixins.

**abstract property CONNECT\_URI\_HELP: str**

Example URI to show the user in help and error messages

**abstract property CONNECT\_URI\_PARAMS: List[str]**

List of parameters given in the path of the URI

**abstract query\_table\_schema**(*path: Tuple[str, ...]*) → Dict[str, tuple]

Query the table for its schema for table in 'path', and return {column: tuple} where the tuple is (table\_name, col\_name, type\_repr, datetime\_precision?, numeric\_precision?, numeric\_scale?)

**Note:** This method exists instead of `select_table_schema()`, just because not all databases support accessing the schema using a SQL query.

**abstract select\_table\_unique\_columns**(*path: Tuple[str, ...]*) → str

Provide SQL for selecting the names of unique columns in the table

**abstract query\_table\_unique\_columns**(*path: Tuple[str, ...]*) → List[str]

Query the table for its unique columns for table in 'path', and return {column}

**abstract parse\_table\_name**(*name: str*) → Tuple[str, ...]

Parse the given table name into a DbPath

**abstract close**()

Close connection(s) to the database instance. Querying will stop functioning.

**abstract property is\_autocommit: bool**

Return whether the database autocommits changes. When false, COMMIT statements are skipped.

**class** skeleton.abcs.database\_types.**AbstractTable**

**abstract select**(\*exprs, distinct=False, \*\*named\_exprs) → *AbstractTable*

Choose new columns, based on the old ones. (aka Projection)

**Parameters**

- **exprs** – List of expressions to constitute the columns of the new table. If not provided, returns all columns in source table (i.e. `select *`)

- **distinct** – ‘select’ or ‘select distinct’
- **named\_exprs** – More expressions to constitute the columns of the new table, aliased to keyword name.

**abstract where**(\*exprs) → *AbstractTable*

Filter the rows, based on the given predicates. (aka Selection)

**abstract order\_by**(\*exprs) → *AbstractTable*

Order the rows lexicographically, according to the given expressions.

**abstract limit**(limit: int) → *AbstractTable*

Stop yielding rows after the given limit. i.e. take the first ‘n=limit’ rows

**abstract join**(target) → *AbstractTable*

Join the current table with the target table, returning a new table containing both side-by-side.

When joining, it’s recommended to use explicit tables names, instead of *this*, in order to avoid potential name collisions.

### Example

```
person = table('person')
city = table('city')

name_and_city = (
    person
    .join(city)
    .on(person['city_id'] == city['id'])
    .select(person['id'], city['name'])
)
```

**abstract group\_by**(\*keys)

Behaves like in SQL, except for a small change in syntax:

A call to *.agg()* must follow every call to *.group\_by()*.

### Example

```
# SELECT a, sum(b) FROM tmp GROUP BY 1
table('tmp').group_by(this.a).agg(this.b.sum())

# SELECT a, sum(b) FROM a GROUP BY 1 HAVING (b > 10)
(table('tmp')
    .group_by(this.a)
    .agg(this.b.sum())
    .having(this.b > 10)
)
```

**abstract count**() → int

SELECT count() FROM self

**abstract union**(other: *ITable*)

SELECT \* FROM self UNION other

**abstract union\_all**(*other: ITable*)

SELECT \* FROM self UNION ALL other

**abstract minus**(*other: ITable*)

SELECT \* FROM self EXCEPT other

**abstract intersect**(*other: ITable*)

SELECT \* FROM self INTERSECT other

**class** `skeleton.abcs.mixins.AbstractMixin`

A mixin for a database dialect

**class** `skeleton.abcs.mixins.AbstractMixin_NormalizeValue`

**abstract normalize\_timestamp**(*value: str, coltype: TemporalType*) → str

Creates an SQL expression, that converts ‘value’ to a normalized timestamp.

The returned expression must accept any SQL datetime/timestamp, and return a string.

Date format: YYYY-MM-DD HH:mm:ss.FFFFFFF

Precision of dates should be rounded up/down according to `coltype.rounds`

**abstract normalize\_number**(*value: str, coltype: FractionalType*) → str

Creates an SQL expression, that converts ‘value’ to a normalized number.

The returned expression must accept any SQL int/numeric/float, and return a string.

Floats/Decimals are expected in the format “I.P”

Where I is the integer part of the number (as many digits as necessary), and must be at least one digit (0). P is the fractional digits, the amount of which is specified with `coltype.precision`. Trailing zeroes may be necessary. If P is 0, the dot is omitted.

Note: We use ‘precision’ differently than most databases. For decimals, it’s the same as `numeric_scale`, and for floats, who use binary precision, it can be calculated as `log10(2**numeric_precision)`.

**normalize\_boolean**(*value: str, \_coltype: Boolean*) → str

Creates an SQL expression, that converts ‘value’ to either ‘0’ or ‘1’.

**normalize\_uuid**(*value: str, coltype: ColType\_UUID*) → str

Creates an SQL expression, that strips uuids of artifacts like whitespace.

**normalize\_value\_by\_type**(*value: str, coltype: ColType*) → str

Creates an SQL expression, that converts ‘value’ to a normalized representation.

The returned expression must accept any SQL value, and return a string.

The default implementation dispatches to a method according to *coltype*:

```
TemporalType    -> normalize_timestamp()
FractionalType  -> normalize_number()
*else*          -> to_string()

(`Integer` falls in the *else* category)
```

**class** `skeleton.abcs.mixins.AbstractMixin_MD5`

Methods for calculating an MD6 hash as an integer.

**abstract md5\_as\_int**(*s: str*) → str

Provide SQL for computing md5 and returning an int

**class** sqeleton.abcs.mixins.**AbstractMixin\_Schema**

Methods for querying the database schema

TODO: Move AbstractDatabase.query\_table\_schema() and friends over here

**table\_information**() → Compilable

Query to return a table of schema information about existing tables

**abstract list\_tables**(*table\_schema: str, like: Optional[Compilable] = None*) → Compilable

Query to select the list of tables in the schema. (query return type: table[str])

If 'like' is specified, the value is applied to the table name, using the 'like' operator.

**class** sqeleton.abcs.mixins.**AbstractMixin\_Regex**

**abstract test\_regex**(*string: Compilable, pattern: Compilable*) → Compilable

Tests whether the regex pattern matches the string. Returns a bool expression.

**class** sqeleton.abcs.mixins.**AbstractMixin\_RandomSample**

**abstract random\_sample\_n**(*tbl: str, size: int*) → str

Take a random sample of the given size, i.e. return 'size' amount of rows

**abstract random\_sample\_ratio\_approx**(*tbl: str, ratio: float*) → str

Take a random sample of the approximate size determined by the ratio (0..1), where 0 means no rows, and 1 means all rows

i.e. the actual mount of rows returned may vary by standard deviation.

**class** sqeleton.abcs.mixins.**AbstractMixin\_TimeTravel**

**abstract time\_travel**(*table: Compilable, before: bool = False, timestamp: Optional[Compilable] = None, offset: Optional[Compilable] = None, statement: Optional[Compilable] = None*) → Compilable

Selects historical data from a table

#### Parameters

- **querying**(*table* - The name of the table whose history we're) -
- **timestamp**(*timestamp* - A constant) -
- **now**(*offset* - the time 'offset' seconds before) -
- **statement**(*statement* - identifier for) -
- **ID** (e.g. *query*) -

Must specify exactly one of *timestamp*, *offset* or *statement*.

**class** sqeleton.abcs.mixins.**AbstractMixin\_OptimizerHints**

**abstract optimizer\_hints**(*optimizer\_hints: str*) → str

Creates a compatible optimizer\_hints string

#### Parameters

- **hints**(*optimizer\_hints* - string of optimizer) -



## SKELETON

**Skeleton** is a Python library for querying SQL databases.

It consists of -

- A fast and concise query builder, inspired by PyPika and SQLAlchemy
- A modular database interface, with drivers for a long list of SQL databases.

It is comparable to other libraries such as SQLAlchemy or PyPika, in terms of API and intended audience. However there are several notable ways in which it is different.

For more information, [See our README](#)





## RESOURCES

- *Install / Get started*
- *Introduction to Sqeleton*
- *List of supported databases*
- *Connection Editor*
- *Python API Reference*
- Source code (git): <https://github.com/datafold/sqeleton>



## PYTHON MODULE INDEX

### S

- `sqeleton`, [25](#)
- `sqeleton.abcs.database_types`, [36](#)
- `sqeleton.abcs.mixins`, [40](#)
- `sqeleton.databases.base`, [25](#)
- `sqeleton.queries.api`, [27](#)
- `sqeleton.queries.ast_classes`, [30](#)
- `sqeleton.queries.compiler`, [35](#)



## INDEX

### A

`AbstractDatabase` (class in *skeleton.abcs.database\_types*), 38  
`AbstractDialect` (class in *skeleton.abcs.database\_types*), 37  
`AbstractMixin` (class in *skeleton.abcs.mixins*), 40  
`AbstractMixin_MD5` (class in *skeleton.abcs.mixins*), 40  
`AbstractMixin_NormalizeValue` (class in *skeleton.abcs.mixins*), 40  
`AbstractMixin_OptimizerHints` (class in *skeleton.abcs.mixins*), 41  
`AbstractMixin_RandomSample` (class in *skeleton.abcs.mixins*), 41  
`AbstractMixin_Regex` (class in *skeleton.abcs.mixins*), 41  
`AbstractMixin_Schema` (class in *skeleton.abcs.mixins*), 41  
`AbstractMixin_TimeTravel` (class in *skeleton.abcs.mixins*), 41  
`AbstractTable` (class in *skeleton.abcs.database\_types*), 38  
`agg()` (*skeleton.queries.ast\_classes.GroupBy* method), 33  
`Alias` (class in *skeleton.queries.ast\_classes*), 30  
`and_()` (in module *skeleton.queries.api*), 28  
`avg()` (in module *skeleton.queries.api*), 28

### B

`BaseDialect` (class in *skeleton.databases.base*), 25  
`BinBoolOp` (class in *skeleton.queries.ast\_classes*), 32  
`BinOp` (class in *skeleton.queries.ast\_classes*), 32

### C

`CaseWhen` (class in *skeleton.queries.ast\_classes*), 31  
`Cast` (class in *skeleton.queries.ast\_classes*), 34  
`close()` (*skeleton.abcs.database\_types.AbstractDatabase* method), 38  
`close()` (*skeleton.databases.base.Database* method), 27  
`close()` (*skeleton.databases.base.ThreadedDatabase* method), 27  
`coalesce()` (in module *skeleton.queries.api*), 29  
`Code` (class in *skeleton.queries.ast\_classes*), 30

`code()` (in module *skeleton.queries.api*), 29  
`ColType_Alphanum` (class in *skeleton.abcs.database\_types*), 36  
`ColType_UUID` (class in *skeleton.abcs.database\_types*), 36  
`Column` (class in *skeleton.queries.ast\_classes*), 32  
`Commit` (class in *skeleton.queries.ast\_classes*), 35  
`CompileError`, 35  
`Compiler` (class in *skeleton.queries.compiler*), 35  
`Concat` (class in *skeleton.queries.ast\_classes*), 31  
`concat()` (*skeleton.abcs.database\_types.AbstractDialect* method), 37  
`concat()` (*skeleton.databases.base.BaseDialect* method), 26  
`CONNECT_URI_HELP` (*skeleton.abcs.database\_types.AbstractDatabase* property), 38  
`CONNECT_URI_PARAMS` (*skeleton.abcs.database\_types.AbstractDatabase* property), 38  
`ConnectError`, 25  
`ConstantTable` (class in *skeleton.queries.ast\_classes*), 34  
`Count` (class in *skeleton.queries.ast\_classes*), 31  
`count()` (*skeleton.abcs.database\_types.AbstractTable* method), 39  
`count()` (*skeleton.queries.ast\_classes.ITable* method), 30  
`create()` (*skeleton.queries.ast\_classes.TablePath* method), 32  
`create_connection()` (*skeleton.databases.base.ThreadedDatabase* method), 27  
`CreateTable` (class in *skeleton.queries.ast\_classes*), 35  
`Cte` (class in *skeleton.queries.ast\_classes*), 34  
`cte()` (in module *skeleton.queries.api*), 27  
`current_timestamp()` (in module *skeleton.queries.api*), 29  
`current_timestamp()` (*skeleton.abcs.database\_types.AbstractDialect* method), 37  
`current_timestamp()` (*skeleton*

`ton.databases.base.BaseDialect` (class in `sqeleton.databases.base`), 26  
`CurrentTimestamp` (class in `sqeleton.queries.ast_classes`), 35

## D

`Database` (class in `sqeleton.databases.base`), 26  
`Date` (class in `sqeleton.abcs.database_types`), 36  
`Datetime` (class in `sqeleton.abcs.database_types`), 36  
`Decimal` (class in `sqeleton.abcs.database_types`), 36  
`dialect` (`sqeleton.abcs.database_types.AbstractDatabase` property), 38  
`drop()` (`sqeleton.queries.ast_classes.TablePath` method), 32  
`DropTable` (class in `sqeleton.queries.ast_classes`), 35

## E

`else_()` (`sqeleton.queries.ast_classes.CaseWhen` method), 31  
`exists()` (in module `sqeleton.queries.api`), 28  
`Explain` (class in `sqeleton.queries.ast_classes`), 34  
`explain_as_text()` (`sqeleton.abcs.database_types.AbstractDialect` method), 37  
`explain_as_text()` (`sqeleton.databases.base.BaseDialect` method), 26  
`ExprNode` (class in `sqeleton.queries.ast_classes`), 30

## F

`Float` (class in `sqeleton.abcs.database_types`), 36  
`FractionalType` (class in `sqeleton.abcs.database_types`), 36  
`Func` (class in `sqeleton.queries.ast_classes`), 31

## G

`group_by()` (`sqeleton.abcs.database_types.AbstractTable` method), 39  
`group_by()` (`sqeleton.queries.ast_classes.ITable` method), 30  
`GroupBy` (class in `sqeleton.queries.ast_classes`), 33

## H

`having()` (`sqeleton.queries.ast_classes.GroupBy` method), 33

## I

`if_()` (in module `sqeleton.queries.api`), 28  
`IKey` (class in `sqeleton.abcs.database_types`), 36  
`In` (class in `sqeleton.queries.ast_classes`), 34  
`insert_expr()` (`sqeleton.queries.ast_classes.TablePath` method), 33

`insert_row()` (`sqeleton.queries.ast_classes.TablePath` method), 32

`insert_rows()` (`sqeleton.queries.ast_classes.TablePath` method), 32

`InsertToTable` (class in `sqeleton.queries.ast_classes`), 35

`Integer` (class in `sqeleton.abcs.database_types`), 37

`intersect()` (`sqeleton.abcs.database_types.AbstractTable` method), 40

`intersect()` (`sqeleton.queries.ast_classes.ITable` method), 30

`is_autocommit` (`sqeleton.abcs.database_types.AbstractDatabase` property), 38

`is_autocommit` (`sqeleton.databases.base.ThreadedDatabase` property), 27

`is_distinct_from()` (`sqeleton.abcs.database_types.AbstractDialect` method), 37

`is_distinct_from()` (`sqeleton.databases.base.BaseDialect` method), 26

`IsDistinctFrom` (class in `sqeleton.queries.ast_classes`), 31

`ITable` (class in `sqeleton.queries.ast_classes`), 30

## J

`Join` (class in `sqeleton.queries.ast_classes`), 33

`join()` (in module `sqeleton.queries.api`), 27

`join()` (`sqeleton.abcs.database_types.AbstractTable` method), 39

`join()` (`sqeleton.queries.ast_classes.ITable` method), 30

## L

`leftjoin()` (in module `sqeleton.queries.api`), 27

`limit()` (`sqeleton.abcs.database_types.AbstractTable` method), 39

`limit()` (`sqeleton.queries.ast_classes.ITable` method), 30

`list_tables()` (`sqeleton.abcs.mixins.AbstractMixin_Schema` method), 41

`list_tables()` (`sqeleton.databases.base.Mixin_Schema` method), 25

`load_mixins()` (`sqeleton.abcs.database_types.AbstractDatabase` class method), 38

`load_mixins()` (`sqeleton.abcs.database_types.AbstractDialect` class method), 37

`load_mixins()` (`sqeleton.databases.base.BaseDialect` class method), 26

`load_mixins()` (*skeleton.databases.base.Database class method*), 27

## M

`max_()` (*in module skeleton.queries.api*), 28

`md5_as_int()` (*skeleton.abcs.mixins.AbstractMixin\_MD5 method*), 40

`min_()` (*in module skeleton.queries.api*), 28

`minus()` (*skeleton.abcs.database\_types.AbstractTable method*), 40

`minus()` (*skeleton.queries.ast\_classes.ITable method*), 30

`Mixin_OptimizerHints` (*class in skeleton.databases.base*), 25

`Mixin_RandomSample` (*class in skeleton.databases.base*), 25

`Mixin_Schema` (*class in skeleton.databases.base*), 25

`module`

`skeleton`, 25

`skeleton.abcs.database_types`, 36

`skeleton.abcs.mixins`, 40

`skeleton.databases.base`, 25

`skeleton.queries.api`, 27

`skeleton.queries.ast_classes`, 30

`skeleton.queries.compiler`, 35

## N

`name` (*skeleton.abcs.database\_types.AbstractDialect property*), 37

`Native_UUID` (*class in skeleton.abcs.database\_types*), 37

`normalize_boolean()` (*skeleton.abcs.mixins.AbstractMixin\_NormalizeValue method*), 40

`normalize_number()` (*skeleton.abcs.mixins.AbstractMixin\_NormalizeValue method*), 40

`normalize_timestamp()` (*skeleton.abcs.mixins.AbstractMixin\_NormalizeValue method*), 40

`normalize_uuid()` (*skeleton.abcs.mixins.AbstractMixin\_NormalizeValue method*), 40

`normalize_value_by_type()` (*skeleton.abcs.mixins.AbstractMixin\_NormalizeValue method*), 40

`NumericType` (*class in skeleton.abcs.database\_types*), 36

## O

`offset_limit()` (*skeleton.abcs.database\_types.AbstractDialect method*), 37

`offset_limit()` (*skeleton.databases.base.BaseDialect method*), 25

`on()` (*skeleton.queries.ast\_classes.Join method*), 33

`optimizer_hints()` (*skeleton.abcs.mixins.AbstractMixin\_OptimizerHints method*), 41

`optimizer_hints()` (*skeleton.databases.base.Mixin\_OptimizerHints method*), 25

`or_()` (*in module skeleton.queries.api*), 28

`order_by()` (*skeleton.abcs.database\_types.AbstractTable method*), 39

`order_by()` (*skeleton.queries.ast\_classes.ITable method*), 30

`outerjoin()` (*in module skeleton.queries.api*), 27

## P

`Param` (*class in skeleton.queries.ast\_classes*), 35

`parse_table_name()` (*skeleton.abcs.database\_types.AbstractDatabase method*), 38

`parse_table_name()` (*skeleton.databases.base.Database method*), 26

`parse_type()` (*skeleton.abcs.database\_types.AbstractDialect method*), 38

`PrecisionType` (*class in skeleton.abcs.database\_types*), 36

`python_type` (*skeleton.abcs.database\_types.ColType\_Alphanum attribute*), 37

`python_type` (*skeleton.abcs.database\_types.ColType\_UUID attribute*), 36

`python_type` (*skeleton.abcs.database\_types.Decimal property*), 36

`python_type` (*skeleton.abcs.database\_types.Float attribute*), 36

`python_type` (*skeleton.abcs.database\_types.IKey property*), 36

`python_type` (*skeleton.abcs.database\_types.Integer attribute*), 37

`python_type` (*skeleton.abcs.database\_types.StringType attribute*), 36

## Q

`QB_TypeError`, 30

`QB_When` (*class in skeleton.queries.ast\_classes*), 31

`query()` (*skeleton.databases.base.Database method*), 26

`query_table_schema()` (*skeleton.abcs.database\_types.AbstractDatabase method*), 38

`query_table_schema()` (*skeleton.databases.base.Database method*), 26

`query_table_unique_columns()` (*skeleton.abcs.database\_types.AbstractDatabase method*), 38

`query_table_unique_columns()` (*skeleton.databases.base.Database method*), 26

`QueryBuilderError`, 30

QueryError, 25

QueryResult (class in *skeleton.databases.base*), 26

quote() (*skeleton.abcs.database\_types.AbstractDialect* method), 37

## R

Random (class in *skeleton.queries.ast\_classes*), 34

random() (*skeleton.abcs.database\_types.AbstractDialect* method), 37

random() (*skeleton.databases.base.BaseDialect* method), 26

random\_sample\_n() (*skeleton.abcs.mixins.AbstractMixin\_RandomSample* method), 41

random\_sample\_n() (*skeleton.databases.base.Mixin\_RandomSample* method), 25

random\_sample\_ratio\_approx() (*skeleton.abcs.mixins.AbstractMixin\_RandomSample* method), 41

random\_sample\_ratio\_approx() (*skeleton.databases.base.Mixin\_RandomSample* method), 25

returning() (*skeleton.queries.ast\_classes.InsertToTable* method), 35

rightjoin() (in module *skeleton.queries.api*), 27

Root (class in *skeleton.queries.compiler*), 35

ROUNDS\_ON\_PREC\_LOSS (*skeleton.abcs.database\_types.AbstractDialect* property), 37

## S

Select (class in *skeleton.queries.ast\_classes*), 34

select() (*skeleton.abcs.database\_types.AbstractTable* method), 38

select() (*skeleton.queries.ast\_classes.ITable* method), 30

select() (*skeleton.queries.ast\_classes.Join* method), 33

select\_table\_schema() (*skeleton.databases.base.Database* method), 26

select\_table\_unique\_columns() (*skeleton.abcs.database\_types.AbstractDatabase* method), 38

select\_table\_unique\_columns() (*skeleton.databases.base.Database* method), 26

set\_timezone\_to\_utc() (*skeleton.abcs.database\_types.AbstractDialect* method), 38

skeleton  
module, 25

skeleton.abcs.database\_types  
module, 36

skeleton.abcs.mixins  
module, 40

skeleton.databases.base  
module, 25

skeleton.queries.api  
module, 27

skeleton.queries.ast\_classes  
module, 30

skeleton.queries.compiler  
module, 35

Statement (class in *skeleton.queries.ast\_classes*), 35

String\_Alphanum (class in *skeleton.abcs.database\_types*), 37

String\_FixedAlphanum (class in *skeleton.abcs.database\_types*), 37

String\_UUID (class in *skeleton.abcs.database\_types*), 37

String\_VaryingAlphanum (class in *skeleton.abcs.database\_types*), 37

StringType (class in *skeleton.abcs.database\_types*), 36

sum\_() (in module *skeleton.queries.api*), 28

## T

table() (in module *skeleton.queries.api*), 28

table\_information() (*skeleton.abcs.mixins.AbstractMixin\_Schema* method), 41

table\_information() (*skeleton.databases.base.Mixin\_Schema* method), 25

TableAlias (class in *skeleton.queries.ast\_classes*), 33

TableOp (class in *skeleton.queries.ast\_classes*), 34

TablePath (class in *skeleton.queries.ast\_classes*), 32

TemporalType (class in *skeleton.abcs.database\_types*), 36

test\_regex() (*skeleton.abcs.mixins.AbstractMixin\_Regex* method), 41

TestRegex (class in *skeleton.queries.ast\_classes*), 31

Text (class in *skeleton.abcs.database\_types*), 37

then() (*skeleton.queries.ast\_classes.QB\_When* method), 31

This (class in *skeleton.queries.ast\_classes*), 34

ThreadedDatabase (class in *skeleton.databases.base*), 27

ThreadLocalInterpreter (class in *skeleton.databases.base*), 25

time\_travel() (*skeleton.abcs.mixins.AbstractMixin\_TimeTravel* method), 41

time\_travel() (*skeleton.queries.ast\_classes.TablePath* method), 33

Timestamp (class in *skeleton.abcs.database\_types*), 36

timestamp\_value() (*skeleton.abcs.database\_types.AbstractDialect* method), 38

timestamp\_value() (*skeleton.databases.base.BaseDialect* method),



26

[TimestampTZ](#) (class in *sqeleton.abcs.database\_types*), 36  
[TimeTravel](#) (class in *sqeleton.queries.ast\_classes*), 35  
[to\\_string\(\)](#) (*sqeleton.abcs.database\_types.AbstractDialect* method), 37  
[truncate\(\)](#) (*sqeleton.queries.ast\_classes.TablePath* method), 32  
[TruncateTable](#) (class in *sqeleton.queries.ast\_classes*), 35  
[type](#) (*sqeleton.queries.ast\_classes.BinBoolOp* attribute), 32  
[type](#) (*sqeleton.queries.ast\_classes.Count* attribute), 31  
[type](#) (*sqeleton.queries.ast\_classes.CurrentTimestamp* attribute), 35  
[type](#) (*sqeleton.queries.ast\_classes.Explain* attribute), 35  
[type](#) (*sqeleton.queries.ast\_classes.In* attribute), 34  
[type](#) (*sqeleton.queries.ast\_classes.IsDistinctFrom* attribute), 31  
[type](#) (*sqeleton.queries.ast\_classes.Random* attribute), 34

## U

[UnaryOp](#) (class in *sqeleton.queries.ast\_classes*), 32  
[union\(\)](#) (*sqeleton.abcs.database\_types.AbstractTable* method), 39  
[union\(\)](#) (*sqeleton.queries.ast\_classes.ITable* method), 30  
[union\\_all\(\)](#) (*sqeleton.abcs.database\_types.AbstractTable* method), 39  
[union\\_all\(\)](#) (*sqeleton.queries.ast\_classes.ITable* method), 30  
[UnknownColType](#) (class in *sqeleton.abcs.database\_types*), 37

## W

[when\(\)](#) (in module *sqeleton.queries.api*), 28  
[when\(\)](#) (*sqeleton.queries.ast\_classes.CaseWhen* method), 31  
[WhenThen](#) (class in *sqeleton.queries.ast\_classes*), 31  
[where\(\)](#) (*sqeleton.abcs.database\_types.AbstractTable* method), 39  
[where\(\)](#) (*sqeleton.queries.ast\_classes.ITable* method), 30