pypika Documentation

Release 0.35.16

Timothy Heys

Nov 26, 2019

Contents

CHAPTER 1

Abstract

What is *PyPika*?

PyPika is a Python API for building SQL queries. The motivation behind *PyPika* is to provide a simple interface for building SQL queries without limiting the flexibility of handwritten SQL. Designed with data analysis in mind, *PyPika* leverages the builder design pattern to construct queries to avoid messy string formatting and concatenation. It is also easily extended to take full advantage of specific features of SQL database vendors.

1.1 What are the design goals for *PyPika*?

PyPika is a fast, expressive and flexible way to replace handwritten SQL (or even ORM for the courageous souls amongst you). Validation of SQL correctness is not an explicit goal of *PyPika*. With such a large number of SQL database vendors providing a robust validation of input data is difficult. Instead you are encouraged to check inputs you provide to *PyPika* or appropriately handle errors raised from your SQL database - just as you would have if you were writing SQL yourself.

CHAPTER 2

Contents

2.1 Installation

PyPika supports python `3.5+`. It may also work on pypy, cython, and jython, but is not being tested for these versions.

To install *PyPika* run the following command:

```
pip install pypika
```

2.2 Tutorial

The main classes in pypika are `pypika.Query`, `pypika.Table`, and `pypika.Field`.

```
from pypika import Query, Table, Field
```

2.2.1 Selecting Data

The entry point for building queries is `pypika.Query`. In order to select columns from a table, the table must first be added to the query. For simple queries with only one table, tables and columns can be references using strings. For more sophisticated queries a `pypika.Table` must be used.

```
q = Query.from_('customers').select('id', 'fname', 'lname', 'phone')
```

To convert the query into raw SQL, it can be cast to a string.

```
str(q)
```

Alternatively, you can use the *Query.get_sql()* function:

```
q.get_sql()
```

2.2.2 Tables, Columns, Schemas, and Databases

In simple queries like the above example, columns in the "from" table can be referenced by passing string names into the `select` query builder function. In more complex examples, the `pypika.Table` class should be used. Columns can be referenced as attributes on instances of `pypika.Table`.

```
from pypika import Table, Query

customers = Table('customers')
q = Query.from_(customers).select(customers.id, customers.fname, customers.lname,␣
↪customers.phone)
```

Both of the above examples result in the following SQL:

```
SELECT id,fname,lname,phone FROM customers
```

An alias for the table can be given using the `.as_` function on `pypika.Table`

```
Table('x_view_customers').as_('customers')
q = Query.from_(customers).select(customers.id, customers.phone)
```

```
SELECT id,phone FROM x_view_customers customers
```

A schema can also be specified. Tables can be referenced as attributes on the schema.

```
from pypika import Table, Query, Schema

views = Schema('views')
q = Query.from_(views.customers).select(customers.id, customers.phone)
```

```
SELECT id,phone FROM views.customers
```

Also references to databases can be used. Schemas can be referenced as attributes on the database.

```
from pypika import Table, Query, Database

my_db = Database('my_db')
q = Query.from_(my_db.analytics.customers).select(customers.id, customers.phone)
```

```
SELECT id,phone FROM my_db.analytics.customers
```

Results can be ordered by using the following syntax:

```
from pypika import Order
Query.from_('customers').select('id', 'fname', 'lname', 'phone').orderby('id',␣
↪order=Order.desc)
```

This results in the following SQL:

```
SELECT "id","fname","lname","phone" FROM "customers" ORDER BY "id" DESC
```

Arithmetic

Arithmetic expressions can also be constructed using pypika. Operators such as +, -, *, and / are implemented by `pypika.Field` which can be used simply with a `pypika.Table` or directly.

```
from pypika import Field

q = Query.from_('account').select(
    Field('revenue') - Field('cost')
)
```

```
SELECT revenue-cost FROM accounts
```

Using `pypika.Table`

```
accounts = Table('accounts')
q = Query.from_(accounts).select(
    accounts.revenue - accounts.cost
)
```

```
SELECT revenue-cost FROM accounts
```

An alias can also be used for fields and expressions.

```
q = Query.from_(accounts).select(
    (accounts.revenue - accounts.cost).as_('profit')
)
```

```
SELECT revenue-cost profit FROM accounts
```

More arithmetic examples

```
table = Table('table')
q = Query.from_(table).select(
    table.foo + table.bar,
    table.foo - table.bar,
    table.foo * table.bar,
    table.foo / table.bar,
    (table.foo+table.bar) / table.fiz,
)
```

```
SELECT foo+bar,foo-bar,foo*bar,foo/bar,(foo+bar)/fiz FROM table
```

Filtering

Queries can be filtered with `pypika.Criterion` by using equality or inequality operators

```
customers = Table('customers')
q = Query.from_(customers).select(
    customers.id, customers.fname, customers.lname, customers.phone
).where(
    customers.lname == 'Mustermann'
)
```

```
SELECT id,fname,lname,phone FROM customers WHERE lname='Mustermann'
```

Query methods such as select, where, groupby, and orderby can be called multiple times. Multiple calls to the where method will add additional conditions as

```
customers = Table('customers')
q = Query.from_(customers).select(
    customers.id, customers.fname, customers.lname, customers.phone
).where(
    customers.fname == 'Max'
).where(
    customers.lname == 'Mustermann'
)
```

```
SELECT id,fname,lname,phone FROM customers WHERE fname='Max' AND lname='Mustermann'
```

Filters such as IN and BETWEEN are also supported

```
customers = Table('customers')
q = Query.from_(customers).select(
    customers.id,customers.fname
).where(
    customers.age[18:65] & customers.status.isin(['new', 'active'])
)
```

```
SELECT id,fname FROM customers WHERE age BETWEEN 18 AND 65 AND status IN ('new',
↪'active')
```

Filtering with complex criteria can be created using boolean symbols `&`, `|`, and `^`.

AND

```
customers = Table('customers')
q = Query.from_(customers).select(
    customers.id, customers.fname, customers.lname, customers.phone
).where(
    (customers.age >= 18) & (customers.lname == 'Mustermann')
)
```

```
SELECT id,fname,lname,phone FROM customers WHERE age>=18 AND lname='Mustermann'
```

OR

```
customers = Table('customers')
q = Query.from_(customers).select(
    customers.id, customers.fname, customers.lname, customers.phone
).where(
    (customers.age >= 18) | (customers.lname == 'Mustermann')
)
```

```
SELECT id,fname,lname,phone FROM customers WHERE age>=18 OR lname='Mustermann'
```

XOR

```
customers = Table('customers')
q = Query.from_(customers).select(
```

(continues on next page)

(continued from previous page)

```
    customers.id, customers.fname, customers.lname, customers.phone
).where(
    (customers.age >= 18) ^ customers.is_registered
)
```

```
SELECT id,fname,lname,phone FROM customers WHERE age>=18 XOR is_registered
```

Convenience Methods

In the *Criterion* class, there are the static methods *any* and *all* that allow building chains AND and OR expressions with a list of terms.

```
from pypika import Criterion

customers = Table('customers')
q = Query.from_(customers).select(
    customers.id,
    customers.fname
).where(
    Criterion.all([
        customers.is_registered,
        customers.age >= 18,
        customers.lname == "Jones",
    ])
)
```

```
SELECT id,fname FROM customers WHERE is_registered AND age>=18 AND lname = "Jones"
```

Grouping and Aggregating

Grouping allows for aggregated results and works similar to `SELECT` clauses.

```
from pypika import functions as fn

customers = Table('customers')
q = Query \
    .from_(customers) \
    .where(customers.age >= 18) \
    .groupby(customers.id) \
    .select(customers.id, fn.Sum(customers.revenue))
```

```
SELECT id,SUM("revenue") FROM "customers" WHERE "age">=18 GROUP BY "id"
```

After adding a `GROUP BY` clause to a query, the `HAVING` clause becomes available. The method `Query.having()` takes a `Criterion` parameter similar to the method `Query.where()`.

```
from pypika import functions as fn

payments = Table('payments')
q = Query \
    .from_(payments) \
    .where(payments.transacted[date(2015, 1, 1):date(2016, 1, 1)]) \
```

(continues on next page)

(continued from previous page)

```
    .groupby(payments.customer_id) \
    .having(fn.Sum(payments.total) >= 1000) \
    .select(payments.customer_id, fn.Sum(payments.total))
```

```
SELECT customer_id,SUM(total) FROM payments
WHERE transacted BETWEEN '2015-01-01' AND '2016-01-01'
GROUP BY customer_id HAVING SUM(total)>=1000
```

Joining Tables and Subqueries

Tables and subqueries can be joined to any query using the `Query.join()` method. Joins can be performed with either a `USING` or `ON` clauses. The `USING` clause can be used when both tables/subqueries contain the same field and the `ON` clause can be used with a criterion. To perform a join, `...join()` can be chained but then must be followed immediately by `...on(<criterion>)` or `...using(*field)`.

Join Types

All join types are supported by *PyPika*.

```
Query \
    .from_(base_table)
    ...
    .join(join_table, JoinType.left)
    ...
```

```
Query \
    .from_(base_table)
    ...
    .left_join(join_table) \
    .right_join(join_table) \
    .inner_join(join_table) \
    .outer_join(join_table) \
    .cross_join(join_table) \
    ...
```

See the list of join types here `pypika.enums.JoinTypes`

Example of a join using *ON*

```
history, customers = Tables('history', 'customers')
q = Query \
    .from_(history) \
    .join(customers) \
    .on(history.customer_id == customers.id) \
    .select(history.star) \
    .where(customers.id == 5)
```

```
SELECT "history".* FROM "history" JOIN "customers" ON "history"."customer_id"=
↪"customers"."id" WHERE "customers"."id"=5
```

As a shortcut, the `Query.join().on_field()` function is provided for joining the (first) table in the `FROM` clause with the joined table when the field name(s) are the same in both tables.

Example of a join using *ON*

```
history, customers = Tables('history', 'customers')
q = Query \
    .from_(history) \
    .join(customers) \
    .on_field('customer_id', 'group') \
    .select(history.star) \
    .where(customers.group == 'A')
```

```
SELECT "history".* FROM "history" JOIN "customers" ON "history"."customer_id"=
↪"customers"."customer_id" AND "history"."group"="customers"."group" WHERE "customers
↪"."group"='A'
```

Example of a join using *USING*

```
history, customers = Tables('history', 'customers')
q = Query \
    .from_(history) \
    .join(customers) \
    .using('customer_id') \
    .select(history.star) \
    .where(customers.id == 5)
```

```
SELECT "history".* FROM "history" JOIN "customers" USING "customer_id" WHERE
↪"customers"."id"=5
```

Example of a correlated subquery in the *SELECT*

```
history, customers = Tables('history', 'customers')
last_purchase_at = Query.from_(history).select(
    history.purchase_at
).where(history.customer_id==customers.customer_id).orderby(
    history.purchase_at, order=Order.desc
).limit(1)
q = Query.from_(customers).select(
    customers.id, last_purchase_at._as('last_purchase_at')
)
```

```
SELECT
  "id",
  (SELECT "history"."purchase_at"
   FROM "history"
   WHERE "history"."customer_id" = "customers"."customer_id"
   ORDER BY "history"."purchase_at" DESC
   LIMIT 1) "last_purchase_at"
FROM "customers"
```

Unions

Both `UNION` and `UNION ALL` are supported. `UNION DISTINCT` is synonomous with "UNION`` so and *PyPika* does not provide a separate function for it. Unions require that queries have the same number of `SELECT` clauses so trying to cast a unioned query to string with through a `UnionException` if the column sizes are mismatched.

To create a union query, use either the `Query.union()` method or + operator with two query instances. For a union all, use `Query.union_all()` or the * operator.

```
provider_a, provider_b = Tables('provider_a', 'provider_b')
q = Query.from_(provider_a).select(
    provider_a.created_time, provider_a.foo, provider_a.bar
) + Query.from_(provider_b).select(
    provider_b.created_time, provider_b.fiz, provider_b.buz
)
```

```
SELECT "created_time","foo","bar" FROM "provider_a" UNION SELECT "created_time","fiz",
↪"buz" FROM "provider_b"
```

Date, Time, and Intervals

Using `pypika.Interval`, queries can be constructed with date arithmetic. Any combination of intervals can be used except for weeks and quarters, which must be used separately and will ignore any other values if selected.

```
from pypika import functions as fn

fruits = Tables('fruits')
q = Query.from_(fruits) \
    .select(fruits.id, fruits.name) \
    .where(fruits.harvest_date + Interval(months=1) < fn.Now())
```

```
SELECT id,name FROM fruits WHERE harvest_date+INTERVAL 1 MONTH<NOW()
```

Tuples

Tuples are supported through the class `pypika.Tuple` but also through the native python tuple wherever possible. Tuples can be used with `pypika.Criterion` in **WHERE** clauses for pairwise comparisons.

```
from pypika import Query, Tuple

q = Query.from_(self.table_abc) \
    .select(self.table_abc.foo, self.table_abc.bar) \
    .where(Tuple(self.table_abc.foo, self.table_abc.bar) == Tuple(1, 2))
```

```
SELECT "foo","bar" FROM "abc" WHERE ("foo","bar")=(1,2)
```

Using `pypika.Tuple` on both sides of the comparison is redundant and *PyPika* supports native python tuples.

```
from pypika import Query, Tuple

q = Query.from_(self.table_abc) \
    .select(self.table_abc.foo, self.table_abc.bar) \
    .where(Tuple(self.table_abc.foo, self.table_abc.bar) == (1, 2))
```

```
SELECT "foo","bar" FROM "abc" WHERE ("foo","bar")=(1,2)
```

Tuples can be used in **IN** clauses.

```
Query.from_(self.table_abc) \
        .select(self.table_abc.foo, self.table_abc.bar) \
        .where(Tuple(self.table_abc.foo, self.table_abc.bar).isin([(1, 1), (2, 2), (3,
↪ 3)]))
```

```
SELECT "foo","bar" FROM "abc" WHERE ("foo","bar") IN ((1,1),(2,2),(3,3))
```

Strings Functions

There are several string operations and function wrappers included in *PyPika*. Function wrappers can be found in the `pypika.functions` package. In addition, *LIKE* and *REGEX* queries are supported as well.

```
from pypika import functions as fn

customers = Tables('customers')
q = Query.from_(customers).select(
    customers.id,
    customers.fname,
    customers.lname,
).where(
    customers.lname.like('Mc%')
)
```

```
SELECT id,fname,lname FROM customers WHERE lname LIKE 'Mc%'
```

```
from pypika import functions as fn

customers = Tables('customers')
q = Query.from_(customers).select(
    customers.id,
    customers.fname,
    customers.lname,
).where(
    customers.lname.regex(r'^[abc][a-zA-Z]+&')
)
```

```
SELECT id,fname,lname FROM customers WHERE lname REGEX '^[abc][a-zA-Z]+&';
```

```
from pypika import functions as fn

customers = Tables('customers')
q = Query.from_(customers).select(
    customers.id,
    fn.Concat(customers.fname, ' ', customers.lname).as_('full_name'),
)
```

```
SELECT id,CONCAT(fname, ' ', lname) full_name FROM customers
```

Custom Functions

Custom Functions allows us to use any function on queries, as some functions are not covered by PyPika as default, we can appeal to Custom functions.

```
from pypika import CustomFunction

customers = Tables('customers')
DateDiff = CustomFunction('DATE_DIFF', ['interval', 'start_date', 'end_date'])

q = Query.from_(customers).select(
    customers.id,
    customers.fname,
    customers.lname,
    DateDiff('day', customers.created_date, customers.updated_date)
)
```

```
SELECT id,fname,lname,DATE_DIFF('day',created_date,updated_date) FROM customers
```

Case Statements

Case statements allow fow a number of conditions to be checked sequentially and return a value for the first condition met or otherwise a default value. The Case object can be used to chain conditions together along with their output using the `when` method and to set the default value using `else_`.

```
from pypika import Case, functions as fn

customers = Tables('customers')
q = Query.from_(customers).select(
    customers.id,
    Case()
        .when(customers.fname == "Tom", "It was Tom")
        .when(customers.fname == "John", "It was John")
        .else_("It was someone else.").as_('who_was_it')
)
```

```
SELECT "id",CASE WHEN "fname"='Tom' THEN 'It was Tom' WHEN "fname"='John' THEN 'It␣
↪was John' ELSE 'It was someone else.' END "who_was_it" FROM "customers"
```

With Clause

With clause allows give a sub-query block a name, which can be referenced in several places within the main SQL query. The SQL WITH clause is basically a drop-in replacement to the normal sub-query.

```
from pypika import Table, AliasedQuery, Query

customers = Table('customers')

sub_query = (Query
             .from_(customers)
             .select('*'))

test_query = (Query
             .with_(sub_query, "an_alias")
```

(continues on next page)

(continued from previous page)

```
            .from_(AliasedQuery("an_alias"))
            .select('*'))
```

You can use as much as *.with_()* as you want.

```
WITH an_alias AS (SELECT * FROM "customers") SELECT * FROM an_alias
```

2.2.3 Inserting Data

Data can be inserted into tables either by providing the values in the query or by selecting them through another query.

By default, data can be inserted by providing values for all columns in the order that they are defined in the table.

Insert with values

```
customers = Table('customers')

q = Query.into(customers).insert(1, 'Jane', 'Doe', 'jane@example.com')
```

```
INSERT INTO customers VALUES (1,'Jane','Doe','jane@example.com')
```

```
customers =  Table('customers')

q = customers.insert(1, 'Jane', 'Doe', 'jane@example.com')
```

```
INSERT INTO customers VALUES (1,'Jane','Doe','jane@example.com')
```

Multiple rows of data can be inserted either by chaining the `insert` function or passing multiple tuples as args.

```
customers = Table('customers')

q = Query.into(customers).insert(1, 'Jane', 'Doe', 'jane@example.com').insert(2, 'John
↪', 'Doe', 'john@example.com')
```

```
customers = Table('customers')

q = Query.into(customers).insert((1, 'Jane', 'Doe', 'jane@example.com'),
                                 (2, 'John', 'Doe', 'john@example.com'))
```

Insert with on Duplicate Key Update

```
customers = Table('customers')

q = Query.into(customers)\
    .insert(1, 'Jane', 'Doe', 'jane@example.com')\
    .on_duplicate_key_update(customers.email, Values(customers.email))
```

```
INSERT INTO customers VALUES (1,'Jane','Doe','jane@example.com') ON DUPLICATE KEY␣
↪UPDATE `email`=VALUES(`email`)
```

`.on_duplicate_key_update` works similar to `.set` for updating rows, additionally it provides the `Values` wrapper to update to the value specified in the `INSERT` clause.

Insert from a SELECT Sub-query

```
INSERT INTO customers VALUES (1,'Jane','Doe','jane@example.com'),(2,'John','Doe',
↪'john@example.com')
```

To specify the columns and the order, use the `columns` function.

```
customers = Table('customers')

q = Query.into(customers).columns('id', 'fname', 'lname').insert(1, 'Jane', 'Doe')
```

```
INSERT INTO customers (id,fname,lname) VALUES (1,'Jane','Doe','jane@example.com')
```

Inserting data with a query works the same as querying data with the additional call to the `into` method in the builder chain.

```
customers, customers_backup = Tables('customers', 'customers_backup')

q = Query.into(customers_backup).from_(customers).select('*')
```

```
INSERT INTO customers_backup SELECT * FROM customers
```

```
customers, customers_backup = Tables('customers', 'customers_backup')

q = Query.into(customers_backup).columns('id', 'fname', 'lname')
    .from_(customers).select(customers.id, customers.fname, customers.lname)
```

```
INSERT INTO customers_backup SELECT "id", "fname", "lname" FROM customers
```

The syntax for joining tables is the same as when selecting data

```
customers, orders, orders_backup = Tables('customers', 'orders', 'orders_backup')

q = Query.into(orders_backup).columns('id', 'address', 'customer_fname', 'customer_
↪lname')
    .from_(customers)
    .join(orders).on(orders.customer_id == customers.id)
    .select(orders.id, customers.fname, customers.lname)
```

```
INSERT INTO "orders_backup" ("id","address","customer_fname","customer_lname")
SELECT "orders"."id","customers"."fname","customers"."lname" FROM "customers"
JOIN "orders" ON "orders"."customer_id"="customers"."id"
```

2.2.4 Updating Data

PyPika allows update queries to be constructed with or without where clauses.

```
customers = Table('customers')

```

(continues on next page)

(continued from previous page)

```
Query.update(customers).set(customers.last_login, '2017-01-01 10:00:00')

Query.update(customers).set(customers.lname, 'smith').where(customers.id == 10)
```

```
UPDATE "customers" SET "last_login"='2017-01-01 10:00:00'

UPDATE "customers" SET "lname"='smith' WHERE "id"=10
```

The syntax for joining tables is the same as when selecting data

```
customers, profiles = Tables('customers', 'profiles')

Query.update(customers)
     .join(profiles).on(profiles.customer_id == customers.id)
     .set(customers.lname, profiles.lname)
```

```
UPDATE "customers"
JOIN "profiles" ON "profiles"."customer_id"="customers"."id"
SET "customers"."lname"="profiles"."lname"
```

Using `pypika.Table` alias to perform the update

```
customers = Table('customers')

customers.update()
        .set(customers.lname, 'smith')
        .where(customers.id == 10)
```

```
UPDATE "customers" SET "lname"='smith' WHERE "id"=10
```

Using `limit` for performing update

```
customers = Table('customers')

customers.update()
        .set(customers.lname, 'smith')
        .limit(2)
```

```
UPDATE "customers" SET "lname"='smith' LIMIT 2
```

2.2.5 Parametrized Queries

PyPika allows you to use `Parameter(str)` term as a placeholder for parametrized queries.

```
customers = Table('customers')

q = Query.into(customers).columns('id', 'fname', 'lname')
    .insert(Parameter(':1'), Parameter(':2'), Parameter(':3'))
```

```
INSERT INTO customers (id,fname,lname) VALUES (:1,:2,:3)
```

This allows you to build prepared statements, and/or avoid SQL-injection related risks.

Due to the mix of syntax for parameters, depending on connector/driver, it is required that you specify the parameter token explicitly.

An example of some common SQL parameter styles used in Python drivers are:

PostgreSQL: `$number` OR `%s` + `:name` (depending on driver)

MySQL: `%s`

SQLite: `?`

Vertica: `:name`

Oracle: `:number` + `:name`

MSSQL: `%(name)s` OR `:name` + `:number` (depending on driver)

You can find out what parameter style is needed for DBAPI compliant drivers here: https://www.python.org/dev/peps/pep-0249/#paramstyle or in the DB driver documentation.

2.3 Advanced Query Features

This section covers the range of functions that are not widely standardized across all SQL databases or meet special needs. *PyPika* intends to support as many features across different platforms as possible. If there are any features specific to a certain platform that PyPika does not support, please create a GitHub issue requesting that it be added.

2.3.1 Handling different database platforms

There can sometimes be differences between how database vendors implement SQL in their platform, for example which quote characters are used. To ensure that the correct SQL standard is used for your platform, the platform-specific Query classes can be used.

```
from pypika import MySQLQuery, MSSQLQuery, PostgreSQLQuery, OracleQuery, VerticaQuery
```

You can use these query classes as a drop in replacement for the default `Query` class shown in the other examples. Again, if you encounter any issues specific to a platform, please create a GitHub issue on this repository.

2.3.2 GROUP BY Modifiers

The *ROLLUP* modifier allows for aggregating to higher levels that the given groups, called super-aggregates.

```
from pypika import Rollup, functions as fn

products = Table('products')

query = Query.from_(products) \
    .select(products.id, products.category, fn.Sum(products.price)) \
    .rollup(products.id, products.category)
```

```
SELECT "id","category",SUM("price") FROM "products" GROUP BY ROLLUP("id","category")
```

2.3.3 Pseudo Column

A pseudo-column is an SQL assigned value (pseudo-field) used in the same context as an column, but not stored on disk. The pseudo-column can change from database to database, so here it's possible to define them.

```
from pypika import Query, PseudoColumn

CurrentDate = PseudoColumn('current_date')

Query.from_('products').select(CurrentDate)
```

```
SELECT current_date FROM "products"
```

2.3.4 Analytic Queries

The package `pypika.analytic` contains analytic function wrappers. These can be used in `SELECT` clauses when building queries for databases that support them. Different functions have different arguments but all require some sort of partitioning.

NTILE and RANK

The `NTILE` function requires a constant integer argument while the `RANK` function takes no arguments. clause.

```
from pypika import Query, Table, analytics as an, functions as fn

store_sales_fact, date_dimension = Table('store_sales_fact', schema='store'), Table(
↪'date_dimension')

total_sales = fn.Sum(store_sales_fact.sales_quantity).as_('TOTAL_SALES')
calendar_month_name = date_dimension.calendar_month_name.as_('MONTH')
ntile = an.NTile(4).order_by(total_sales).as_('NTILE')

query = Query.from_(store_sales_fact) \
    .join(date_dimension).using('date_key') \
    .select(calendar_month_name, total_sales, ntile) \
    .groupby(calendar_month_name) \
    .orderby(ntile)
```

```
SELECT "date_dimension"."calendar_month_name" "MONTH",SUM("store_sales_fact"."sales_
↪quantity") "TOTAL_SALES",NTILE(4) OVER(PARTITION BY  ORDER BY SUM("store_sales_fact
↪"."sales_quantity")) "NTILE" FROM "store"."store_sales_fact" JOIN "date_dimension"␣
↪USING ("date_key") GROUP BY "date_dimension"."calendar_month_name" ORDER BY␣
↪NTILE(4) OVER(PARTITION BY  ORDER BY SUM("store_sales_fact"."sales_quantity"))
```

FIRST_VALUE and LAST_VALUE

`FIRST_VALUE` and `LAST_VALUE` both expect a single argument. They also support an additional `IGNORE NULLS` clause.

```
from pypika import Query, Table, analytics as an

t_month = Table('t_month')
```

(continues on next page)

(continued from previous page)

```
first_month = an.FirstValue(t_month.month) \
    .over(t_month.season) \
    .orderby(t_month.id)

last_month = an.LastValue(t_month.month) \
    .over(t_month.season) \
    .orderby(t_month.id) \
    .ignore_nulls()

query = Query.from_(t_month) \
    .select(first_month, last_month)
```

```
SELECT FIRST_VALUE("month") OVER(PARTITION BY "season" ORDER BY "id"),LAST_VALUE(
↪"month" IGNORE NULLS) OVER(PARTITION BY "season" ORDER BY "id") FROM "t_month"
```

MEDIAN, AVG and STDDEV

These functions take one or more arguments

```
from pypika import Query, Table, analytics as an

customer_dimension = Table('customer_dimension')

median_income = an.Median(customer_dimension.annual_income).over(customer_dimension.
↪customer_state).as_('MEDIAN')
avg_income = an.Avg(customer_dimension.annual_income).over(customer_dimension.
↪customer_state).as_('AVG')
stddev_income = an.StdDev(customer_dimension.annual_income).over(customer_dimension.
↪customer_state).as_('STDDEV')

query = Query.from_(customer_dimension) \
    .select(median_income, avg_income, stddev_income) \
    .where(customer_dimension.customer_state.isin(['DC', 'WI'])) \
    .orderby(customer_dimension.customer_state)
```

```
SELECT MEDIAN("annual_income") OVER(PARTITION BY "customer_state") "MEDIAN",AVG(
↪"annual_income") OVER(PARTITION BY "customer_state") "AVG",STDDEV("annual_income")␣
↪OVER(PARTITION BY "customer_state") "STDDEV" FROM "customer_dimension" WHERE
↪"customer_state" IN ('DC','WI') ORDER BY "customer_state"
```

2.4 Window Frames

Functions which use window aggregation expose the functions `rows()` and `range()` with varying parameters to define the window. Both of these functions take one or two parameters which specify the offset boundaries. Boundaries can be set either as the current row with `an.CURRENT_ROW` or a value preceding or following the current row with `an.Preceding(constant_value)` and `an.Following(constant_value)`. The ranges can be unbounded preceding or following the current row by omitting the `constant_value` parameter like `an.Preceding()` or `an.Following()`.

`FIRST_VALUE` and `LAST_VALUE` also support window frames.

```
from pypika import Query, Table, analytics as an

t_transactions = Table('t_customers')

rolling_7_sum = an.Sum(t_transactions.total) \
    .over(t_transactions.item_id) \
    .orderby(t_transactions.day) \
    .rows(an.Preceding(7), an.CURRENT_ROW)

query = Query.from_(t_transactions) \
    .select(rolling_7_sum)
```

```
SELECT SUM("total") OVER(PARTITION BY "item_id" ORDER BY "day" ROWS BETWEEN 7␣
↪PRECEDING AND CURRENT ROW) FROM "t_customers"
```

2.5 Extending PyPika

2.5.1 SQL Functions not included in PyPika

PyPika includes a couple of the most common SQL functions, but due to many differences between different SQL databases, many are not included. Any SQL function can be implemented in PyPika by extended the `pypika.Function` class.

When defining SQL function wrappers, it is necessary to define the name of the SQL function as well as the arguments it requires.

```
from pypika import Function

class CurDate(Function):
    def __init__(self, alias=None):
        super(CurDate, self).__init__('CURRENT_DATE', alias=alias)

q = Query.select(CurDate())
```

```
from pypika import Function

class DateDiff(Function):
    def __init__(self, interval, start_date, end_date, alias=None):
        super(DateDiff, self).__init__('DATEDIFF', interval, start_date, end_date,␣
↪alias=alias)
```

There is also a helper function `pypika.CustomFunction` which enables 1-line creation of a SQL function wrapper.

```
from pypika import CustomFunction

customers = Tables('customers')
DateDiff = CustomFunction('DATE_DIFF', ['interval', 'start_date', 'end_date'])

q = Query.from_(customers).select(
    customers.id,
    customers.fname,
    customers.lname,
```

(continues on next page)

(continued from previous page)

```
    DateDiff('day', customers.created_date, customers.updated_date)
)
```

Similarly analytic functions can be defined by extending `pypika.AnalyticFunction`.

```
from pypika import AnalyticFunction

class RowNumber(AnalyticFunction):
    def __init__(self, **kwargs):
        super(RowNumber, self).__init__('ROW_NUMBER', **kwargs)


expr =

q = Query.from_(self.table_abc) \
        .select(an.RowNumber()
                    .over(self.table_abc.foo)
                    .orderby(self.table_abc.date))
```

```
SELECT ROW_NUMBER() OVER(PARTITION BY "foo" ORDER BY "date") FROM "abc"
```

2.6 API Reference

2.6.1 pypika package

pypika.enums module

class pypika.enums.**Arithmetic**
Bases: enum.Enum

An enumeration.

add = '+'

div = '/'

mul = '*'

sub = '-'

class pypika.enums.**Boolean**
Bases: *pypika.enums.Comparator*

An enumeration.

and_ = 'AND'

false = 'FALSE'

or_ = 'OR'

true = 'TRUE'

xor_ = 'XOR'

class pypika.enums.**Comparator**
Bases: enum.Enum

An enumeration.

class pypika.enums.**DatePart**
Bases: enum.Enum

An enumeration.

day = 'DAY'

hour = 'HOUR'

microsecond = 'MICROSECOND'

minute = 'MINUTE'

month = 'MONTH'

quarter = 'QUARTER'

second = 'SECOND'

week = 'WEEK'

year = 'YEAR'

class pypika.enums.**Dialects**
Bases: enum.Enum

An enumeration.

CLICKHOUSE = 'clickhouse'

MSSQL = 'mssql'

MYSQL = 'mysql'

ORACLE = 'oracle'

POSTGRESQL = 'postgressql'

REDSHIFT = 'redshift'

SNOWFLAKE = 'snowflake'

SQLLITE = 'sqllite'

VERTICA = 'vertica'

class pypika.enums.**Equality**
Bases: *pypika.enums.Comparator*

An enumeration.

eq = '='

gt = '>'

gte = '>='

lt = '<'

lte = '<='

ne = '<>'

class pypika.enums.**JSONOperators**
Bases: enum.Enum

An enumeration.

CONTAINED_BY = '<@'

CONTAINS = '@>'

GET_JSON_VALUE = '->'

GET_PATH_JSON_VALUE = '#>'

GET_PATH_TEXT_VALUE = '#>>'

GET_TEXT_VALUE = '->>'

HAS_ANY_KEYS = '?|'

HAS_KEY = '?'

HAS_KEYS = '?&'

class pypika.enums.**JoinType**
Bases: enum.Enum

An enumeration.

cross = 'CROSS'

full_outer = 'FULL OUTER'

inner = ''

left = 'LEFT'

left_outer = 'LEFT OUTER'

outer = 'FULL OUTER'

right = 'RIGHT'

right_outer = 'RIGHT OUTER'

class pypika.enums.**Matching**
Bases: *pypika.enums.Comparator*

An enumeration.

bin_regex = ' REGEX BINARY '

ilike = ' ILIKE '

like = ' LIKE '

not_ilike = ' NOT ILIKE '

not_like = ' NOT LIKE '

regex = ' REGEX '

class pypika.enums.**Order**
Bases: enum.Enum

An enumeration.

asc = 'ASC'

desc = 'DESC'

class pypika.enums.**SqlType**(*name*)
Bases: object

get_sql(***kwargs*)

class pypika.enums.**SqlTypeLength**(*name*, *length*)
Bases: object

get_sql(***kwargs*)

class pypika.enums.**SqlTypes**
Bases: object

BINARY = <pypika.enums.SqlType object>

BOOLEAN = 'BOOLEAN'

CHAR = <pypika.enums.SqlType object>

DATE = 'DATE'

FLOAT = 'FLOAT'

INTEGER = 'INTEGER'

LONG_VARBINARY = <pypika.enums.SqlType object>

LONG_VARCHAR = <pypika.enums.SqlType object>

NUMERIC = 'NUMERIC'

SIGNED = 'SIGNED'

TIME = 'TIME'

TIMESTAMP = 'TIMESTAMP'

UNSIGNED = 'UNSIGNED'

VARBINARY = <pypika.enums.SqlType object>

VARCHAR = <pypika.enums.SqlType object>

class pypika.enums.**UnionType**
Bases: enum.Enum

An enumeration.

all = ' ALL'

distinct = ''

pypika.functions module

Package for SQL functions wrappers

class pypika.functions.**Abs**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**ApproximatePercentile**(*term*, *percentile*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

get_special_params_sql(***kwargs*)

class pypika.functions.**Ascii**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Avg**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**Bin**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Cast**(*term*, *as_type*, *alias=None*)
Bases: *pypika.terms.Function*

get_special_params_sql(***kwargs*)

class pypika.functions.**Coalesce**(*term*, **default_values*, ***kwargs*)
Bases: *pypika.terms.Function*

class pypika.functions.**Concat**(**terms*, ***kwargs*)
Bases: *pypika.terms.Function*

class pypika.functions.**Convert**(*term*, *encoding*, *alias=None*)
Bases: *pypika.terms.Function*

get_special_params_sql(***kwargs*)

class pypika.functions.**Count**(*param*, *alias=None*)
Bases: *pypika.functions.DistinctOptionFunction*

class pypika.functions.**CurDate**(*alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**CurTime**(*alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**CurTimestamp**(*alias=None*)
Bases: *pypika.terms.Function*

get_function_sql(***kwargs*)

class pypika.functions.**Date**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**DateAdd**(*date_part*, *interval*, *term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**DateDiff**(*interval*, *start_date*, *end_date*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**DistinctOptionFunction**(*name*, **args*, ***kwargs*)
Bases: *pypika.terms.AggregateFunction*

distinct(**args*, ***kwargs*)

get_function_sql(***kwargs*)

class pypika.functions.**Extract**(*date_part*, *field*, *alias=None*)
Bases: *pypika.terms.Function*

get_special_params_sql(***kwargs*)

class pypika.functions.**First**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**Floor**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**IfNull**(*condition*, *term*, ***kwargs*)
Bases: *pypika.terms.Function*

class pypika.functions.**Insert**(*term*, *start*, *stop*, *subterm*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**IsNull**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Last**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**Length**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Lower**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Max**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**Min**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**NVL**(*condition*, *term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Now**(*alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**NullIf**(*term*, *condition*, ***kwargs*)
Bases: *pypika.terms.Function*

class pypika.functions.**RegexpLike**(*term*, *pattern*, *modifiers=None*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**RegexpMatches**(*term*, *pattern*, *modifiers=None*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Reverse**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Signed**(*term*, *alias=None*)
Bases: *pypika.functions.Cast*

class pypika.functions.**SplitPart**(*term*, *delimiter*, *index*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Sqrt**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Std**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**StdDev**(*term*, *alias=None*)
Bases: *pypika.terms.AggregateFunction*

class pypika.functions.**Substring**(*term*, *start*, *stop*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Sum**(*term*, *alias=None*)
Bases: *pypika.functions.DistinctOptionFunction*

class pypika.functions.**TimeDiff**(*start_time*, *end_time*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Timestamp**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**TimestampAdd**(*date_part*, *interval*, *term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**ToChar**(*term*, *as_type*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**ToDate**(*value*, *format_mask*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Trim**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**Unsigned**(*term*, *alias=None*)
Bases: *pypika.functions.Cast*

class pypika.functions.**Upper**(*term*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.functions.**UtcTimestamp**(*alias=None*)
Bases: *pypika.terms.Function*

pypika.queries module

class pypika.queries.**AliasedQuery**(*name*, *query=None*)
Bases: *pypika.queries.Selectable*

get_sql(***kwargs*)

class pypika.queries.**Column**(*column_name*, *column_type=None*)
Bases: object

get_sql(***kwargs*)

class pypika.queries.**CreateQueryBuilder**(*dialect=None*)
Bases: object

Query builder used to build CREATE queries.

ALIAS_QUOTE_CHAR = None

QUOTE_CHAR = '"'

SECONDARY_QUOTE_CHAR = "'"

as_select(**args*, ***kwargs*)

columns(**args*, ***kwargs*)

create_table(**args*, ***kwargs*)

get_sql(***kwargs*)

temporary(**args*, ***kwargs*)

class pypika.queries.**Database**(*name*, *parent=None*)
Bases: *pypika.queries.Schema*

class pypika.queries.**Join**(*item*, *how*)
Bases: object

get_sql(***kwargs*)

replace_table(**args*, ***kwargs*)

validate(*_from*, *_joins*)

class pypika.queries.**JoinOn**(*item*, *how*, *criteria*, *collate=None*)
Bases: *pypika.queries.Join*

get_sql(***kwargs*)

replace_table(**args*, ***kwargs*)

validate(*_from*, *_joins*)

class pypika.queries.**JoinUsing**(*item*, *how*, *fields*)
Bases: *pypika.queries.Join*

get_sql(***kwargs*)

replace_table(**args*, ***kwargs*)

validate(*_from*, *_joins*)

class pypika.queries.**Joiner**(*query*, *item*, *how*, *type_label*)
Bases: object

cross()
Return cross join

on(*criterion*, *collate=None*)

on_field(**fields*)

using(**fields*)

class pypika.queries.**Query**
Bases: object

Query is the primary class and entry point in pypika. It is used to build queries iteratively using the builder design pattern.

This class is immutable.

classmethod create_table(*table*)
Query builder entry point. Initializes query building and sets the table name to be created. When using this function, the query becomes a CREATE statement.

Parameters **table** – An instance of a Table object or a string table name.

Returns CreateQueryBuilder

classmethod from_(*table*)
Query builder entry point. Initializes query building and sets the table to select from. When using this function, the query becomes a SELECT query.

Parameters **table** – Type: Table or str

An instance of a Table object or a string table name.

:returns QueryBuilder

classmethod into(*table*)
Query builder entry point. Initializes query building and sets the table to insert into. When using this function, the query becomes an INSERT query.

Parameters **table** – Type: Table or str

An instance of a Table object or a string table name.

:returns QueryBuilder

classmethod select(**terms*)
Query builder entry point. Initializes query building without a table and selects fields. Useful when testing SQL functions.

Parameters **terms** – Type: list[expression]

A list of terms to select. These can be any type of int, float, str, bool, or Term. They cannot be a Field unless the function `Query.from_` is called first.

:returns QueryBuilder

classmethod update(*table*)
Query builder entry point. Initializes query building and sets the table to update. When using this function, the query becomes an UPDATE query.

Parameters **table** – Type: Table or str

An instance of a Table object or a string table name.

:returns QueryBuilder

classmethod with_(*table*, *name*)

class pypika.queries.**QueryBuilder**(*dialect=None*, *wrap_union_queries=True*, *wrapper_cls=<class 'pypika.terms.ValueWrapper'>*)
Bases: *pypika.queries.Selectable*, *pypika.terms.Term*

Query Builder is the main class in pypika which stores the state of a query and offers functions which allow the state to be branched immutably.

ALIAS_QUOTE_CHAR = None

QUOTE_CHAR = '"'

SECONDARY_QUOTE_CHAR = "'"

columns(**args*, ***kwargs*)

cross_join(*item*)

delete(**args*, ***kwargs*)

distinct(**args*, ***kwargs*)

do_join(*join*)

fields()

force_index(**args*, ***kwargs*)

from_(**args*, ***kwargs*)

get_sql(*with_alias=False*, *subquery=False*, ***kwargs*)

groupby(**args*, ***kwargs*)

having(**args*, ***kwargs*)

ignore(**args*, ***kwargs*)

inner_join(*item*)

insert(**args*, ***kwargs*)

into(**args*, ***kwargs*)

is_joined(*table*)

join(**args*, ***kwargs*)

left_join(*item*)

limit(**args*, ***kwargs*)

offset(**args*, ***kwargs*)

orderby(**args*, ***kwargs*)

outer_join(*item*)

prewhere(**args*, ***kwargs*)

replace(**args*, ***kwargs*)

replace_table(**args*, ***kwargs*)

right_join(*item*)

rollup(**args*, ***kwargs*)

select(**args*, ***kwargs*)

set(**args*, ***kwargs*)

union(**args*, ***kwargs*)

union_all(**args*, ***kwargs*)

update(**args*, ***kwargs*)

where(**args*, ***kwargs*)

with_(**args*, ***kwargs*)

with_totals(**args*, ***kwargs*)

class pypika.queries.**Schema**(*name*, *parent=None*)
Bases: object

get_sql(*quote_char=None*, ***kwargs*)

class pypika.queries.**Selectable**(*alias*)
Bases: object

as_(**args*, ***kwargs*)

field(*name*)

star

class pypika.queries.**Table**(*name*, *schema=None*, *alias=None*)
Bases: *pypika.queries.Selectable*

get_sql(***kwargs*)

insert(**terms*)
Perform an INSERT operation on the current table

Parameters **terms** – Type: list[expression]

A list of terms to select. These can be any type of int, float, str, bool or any other valid data

Returns QueryBuilder

select(**terms*)
Perform a SELECT operation on the current table

Parameters **terms** – Type: list[expression]

A list of terms to select. These can be any type of int, float, str, bool or Term or a Field.

Returns QueryBuilder

update()
Perform an UPDATE operation on the current table

Returns QueryBuilder

pypika.queries.**make_columns**(**names*)
Shortcut to create many columns. If *names* param is a tuple, the first position will refer to the *name* while the second will be its *type*. Any other data structure will be treated as a whole as the *name*.

pypika.queries.**make_tables**(**names*, ***kwargs*)
Shortcut to create many tables. If *names* param is a tuple, the first position will refer to the *_table_name* while the second will be its *alias*. Any other data structure will be treated as a whole as the *_table_name*.

pypika.terms module

class pypika.terms.**AggregateFunction**(*name*, **args*, ***kwargs*)
Bases: *pypika.terms.Function*

is_aggregate = True

class pypika.terms.**AnalyticFunction**(*name*, **args*, ***kwargs*)
Bases: *pypika.terms.Function*

get_function_sql(***kwargs*)

get_partition_sql(***kwargs*)

is_analytic = True

orderby(**args*, ***kwargs*)

over(**args*, ***kwargs*)

class pypika.terms.**ArithmeticExpression**(*operator*, *left*, *right*, *alias=None*)
Bases: *pypika.terms.Term*

Wrapper for an arithmetic function. Can be simple with two terms or complex with nested terms. Order of operations are also preserved.

add_order = [<Arithmetic.add: '+'>, <Arithmetic.sub: '-'>]

fields()

get_sql(*with_alias=False*, ***kwargs*)

is_aggregate
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.

mul_order = [<Arithmetic.mul: '*'>, <Arithmetic.div: '/'>]

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**Array**(**values*)
Bases: *pypika.terms.Tuple*

get_sql(***kwargs*)

class pypika.terms.**BasicCriterion**(*comparator*, *left*, *right*, *alias=None*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(*quote_char='"'*, *with_alias=False*, ***kwargs*)

is_aggregate
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.

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**BetweenCriterion**(*term*, *start*, *end*, *alias=None*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(***kwargs*)

is_aggregate
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.

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**BitwiseAndCriterion**(*term*, *value*, *alias=None*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(***kwargs*)

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**Bracket**(*term*)
Bases: *pypika.terms.Tuple*

get_sql(***kwargs*)

class pypika.terms.**Case**(*alias=None*)
Bases: *pypika.terms.Term*

else_(**args*, ***kwargs*)

fields()

get_sql(*with_alias=False*, ***kwargs*)

is_aggregate
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.

replace_table(**args*, ***kwargs*)

tables_

when(**args*, ***kwargs*)

class pypika.terms.**ComplexCriterion**(*comparator*, *left*, *right*, *alias=None*)
Bases: *pypika.terms.BasicCriterion*

fields()

get_sql(*subcriterion=False*, ***kwargs*)

needs_brackets(*term*)

class pypika.terms.**ContainsCriterion**(*term*, *container*, *alias=None*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(*subquery=None*, ***kwargs*)

is_aggregate
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.

negate()

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**Criterion**(*alias=None*)
Bases: *pypika.terms.Term*

static all(*terms=()*)

static any(*terms=()*)

fields()

get_sql()

class pypika.terms.**CustomFunction**(*name*, *params=None*)
Bases: object

class pypika.terms.**EmptyCriterion**
Bases: object

is_aggregate = None

tables_ = {}

class pypika.terms.**Field**(*name*, *alias=None*, *table=None*)
Bases: *pypika.terms.Criterion*, *pypika.terms.JSON*

fields()

get_sql(*with_alias=False*, *with_namespace=False*, *quote_char=None*, *secondary_quote_char="'"*, ***kwargs*)

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**Function**(*name*, **args*, ***kwargs*)
Bases: *pypika.terms.Criterion*

fields()

get_function_sql(***kwargs*)

get_special_params_sql(***kwargs*)

get_sql(*with_alias=False*, *with_namespace=False*, *quote_char=None*, *dialect=None*, ***kwargs*)

is_aggregate
This is a shortcut that assumes if a function has a single argument and that argument is aggregated, then this function is also aggregated. A more sophisticated approach is needed, however it is unclear how that might work. :returns:

True if the function accepts one argument and that argument is aggregate.

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**IgnoreNullsAnalyticFunction**(*name*, **args*, ***kwargs*)
Bases: *pypika.terms.AnalyticFunction*

get_special_params_sql(***kwargs*)

ignore_nulls(**args*, ***kwargs*)

class pypika.terms.**Index**(*name*, *alias=None*)
Bases: *pypika.terms.Term*

get_sql(*quote_char=None*, ***kwargs*)

class pypika.terms.**Interval**(*years=0*, *months=0*, *days=0*, *hours=0*, *minutes=0*, *seconds=0*, *microseconds=0*, *quarters=0*, *weeks=0*, *dialect=None*)
Bases: object

fields()

get_sql(***kwargs*)

labels = ['YEAR', 'MONTH', 'DAY', 'HOUR', 'MINUTE', 'SECOND', 'MICROSECOND']

tables_

templates = {<Dialects.MYSQL: 'mysql'>: 'INTERVAL {expr} {unit}', <Dialects.POSTGRESQ

trim_pattern = re.compile('(^0+\\.)|(\\.0+$)|(^[0\\-.:]+[\\-:])|([\\-:.][0\\-.:

units = ['years', 'months', 'days', 'hours', 'minutes', 'seconds', 'microseconds']

class pypika.terms.**JSON**(*value*, *alias=None*)
Bases: *pypika.terms.Term*

contained_by(*other*)

contains(*other*)

get_json_value(*key_or_index: Union[str, int]*)

get_path_json_value(*path_json: str*)

get_path_text_value(*path_json: str*)

get_sql(*secondary_quote_char="'"*, ***kwargs*)

get_text_value(*key_or_index: Union[str, int]*)

has_any_keys(*other: Iterable[T_co]*)

has_key(*other*)

has_keys(*other: Iterable[T_co]*)

table = None

class pypika.terms.**Mod**(*term*, *modulus*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.terms.**Negative**(*term*)
Bases: *pypika.terms.Term*

get_sql(***kwargs*)

is_aggregate
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.

class pypika.terms.**NestedCriterion**(*comparator*, *nested_comparator*, *left*, *right*, *nested*, *alias=None*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(*with_alias=False*, ***kwargs*)

is_aggregate
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.

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**Not**(*term*, *alias=None*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(***kwargs*)

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**NullCriterion**(*term*, *alias=None*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(***kwargs*)

replace_table(**args*, ***kwargs*)

tables_

class pypika.terms.**NullValue**(*alias=None*)
Bases: *pypika.terms.Term*

fields()

get_sql(***kwargs*)

class pypika.terms.**Parameter**(*placeholder*)
Bases: *pypika.terms.Term*

fields()

get_sql(***kwargs*)

is_aggregate = None

class pypika.terms.**Pow**(*term*, *exponent*, *alias=None*)
Bases: *pypika.terms.Function*

class pypika.terms.**PseudoColumn**(*name*)
Bases: *pypika.terms.Term*

Represents a pseudo column (a "column" which yields a value when selected but is not actually a real table column).

fields()

get_sql(***kwargs*)

class pypika.terms.**Rollup**(**terms*)
Bases: *pypika.terms.Function*

class pypika.terms.**Star**(*table=None*)
Bases: *pypika.terms.Field*

get_sql(*with_alias=False*, *with_namespace=False*, *quote_char=None*, ***kwargs*)

tables_

class pypika.terms.**Term**(*alias=None*)
Bases: object

as_(**args*, ***kwargs*)

between(*lower*, *upper*)

bin_regex(*pattern*)

bitwiseand(*value*)

eq(*other*)

fields()

get_sql(***kwargs*)

gt(*other*)

gte(*other*)

ilike(*expr*)

is_aggregate = False

isin(*arg*)

isnull()

like(*expr*)

lt(*other*)

lte(*other*)

ne(*other*)

negate()

not_ilike(*expr*)

not_like(*expr*)

notin(*arg*)

notnull()

regex(*pattern*)

replace_table(*current_table*, *new_table*)
Replaces all occurrences of the specified table with the new table. Useful when reusing fields across queries. The base implementation returns self because not all terms have a table property.

Parameters

- **current_table** – The table to be replaced.
- **new_table** – The table to replace with.

Returns Self.

tables_

static wrap_constant(*val*, *wrapper_cls=None*)
Used for wrapping raw inputs such as numbers in Criterions and Operator.

For example, the expression F('abc')+1 stores the integer part in a ValueWrapper object.

Parameters

- **val** – Any value.
- **wrapper_cls** – A pypika class which wraps a constant value so it can be handled as a component of the query.

Returns Raw string, number, or decimal values will be returned in a ValueWrapper. Fields and other parts of the querybuilder will be returned as inputted.

static wrap_json(*val*, *wrapper_cls=None*)

class pypika.terms.**Tuple**(**values*)
Bases: *pypika.terms.Criterion*

fields()

get_sql(***kwargs*)

is_aggregate
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.

replace_table(**args*, ***kwargs*)

class pypika.terms.**ValueWrapper**(*value*, *alias=None*)
Bases: *pypika.terms.Term*

fields()

get_sql(*quote_char=None*, *secondary_quote_char="'"*, ***kwargs*)

get_value_sql(***kwargs*)

is_aggregate = None

class pypika.terms.**Values**(*field*)
Bases: *pypika.terms.Term*

get_sql(*quote_char=None*, ***kwargs*)

class pypika.terms.**WindowFrameAnalyticFunction**(*name*, **args*, ***kwargs*)
Bases: *pypika.terms.AnalyticFunction*

class Edge(*value=None*)
Bases: object

get_frame_sql()

get_partition_sql(***kwargs*)

range(**args*, ***kwargs*)

rows(**args*, ***kwargs*)

pypika.utils module

exception pypika.utils.**CaseException**
Bases: Exception

exception pypika.utils.**DialectNotSupported**
Bases: Exception

exception pypika.utils.**FunctionException**
Bases: Exception

exception pypika.utils.**GroupingException**
Bases: Exception

exception pypika.utils.**JoinException**
Bases: Exception

exception pypika.utils.**QueryException**
Bases: Exception

exception pypika.utils.**RollupException**
Bases: Exception

exception pypika.utils.**UnionException**
Bases: Exception

pypika.utils.**builder**(*func*)
Decorator for wrapper "builder" functions. These are functions on the Query class or other classes used for building queries which mutate the query and return self. To make the build functions immutable, this decorator is used which will deepcopy the current instance. This decorator will return the return value of the inner function or the new copy of the instance. The inner function does not need to return self.

pypika.utils.**format_alias_sql**(*sql*, *alias*, *quote_char=None*, *alias_quote_char=None*, ***kwargs*)

pypika.utils.**format_quotes**(*value*, *quote_char*)

pypika.utils.**ignore_copy**(*func*)
Decorator for wrapping the __getattr__ function for classes that are copied via deepcopy. This prevents infinite recursion caused by deepcopy looking for magic functions in the class. Any class implementing __getattr__ that is meant to be deepcopy'd should use this decorator.

deepcopy is used by pypika in builder functions (decorated by @builder) to make the results immutable. Any data model type class (stored in the Query instance) is copied.

pypika.utils.**resolve_is_aggregate**(*values*)
Resolves the is_aggregate flag for an expression that contains multiple terms. This works like a voter system, each term votes True or False or abstains with None.

Parameters **values** – A list of booleans (or None) for each term in the expression

Returns If all values are True or None, True is returned. If all values are None, None is returned. Otherwise, False is returned.

pypika.utils.**validate**(**args*, *exc=None*, *type=None*)

Module contents

PyPika is divided into a couple of modules, primarily the `queries` and `terms` modules.

pypika.queries

This is where the `Query` class can be found which is the core class in PyPika. Also, other top level classes such as `Table` can be found here. `Query` is a container that holds all of the `Term` types together and also serializes the builder to a string.

pypika.terms

This module contains the classes which represent individual parts of queries that extend the `Term` base class.

pypika.functions

Wrappers for common SQL functions are stored in this package.

pypika.enums

Enumerated values are kept in this package which are used as options for Queries and Terms.

pypika.utils

This contains all of the utility classes such as exceptions and decorators.

CHAPTER 3

Indices and tables

- genindex
- modindex

CHAPTER 4

License

Crafted with in Berlin.

Python Module Index

p

Index

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y