pypika Documentation

Release 0.0.1

Timothy Heys

Dec 12, 2017

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.

CHAPTER 2

Contents

2.1 Installation

PyPika supports python `2.7` and `3.3+`. 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)
```

Using `pypika.Table`

```
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
```

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(
    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
```

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 ORDER BY id ASC
```

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)]
).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)`.

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
).on(
    'customer_id'
).select(
    history.star
).where(
    customers.id == 5
)
```

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

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
```

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"
```

2.2.2 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')
```

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 a SELECT 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
```

2.2.3 Updating Data

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

```
customers = Table('customers')

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

Query.update(customers).set('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
```

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 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')

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

PyPika can be extended to include additional features that are not included.

Adding functions can be achieved by extending `pypika.Function`.

WRITEME

2.6 API Reference

2.6.1 pypika package

pypika.enums module

class `pypika.enums.`**`Arithmetic`**(**args*, ***kwds*)

Bases: `aenum.Enum`

add = <Arithmetic.add: '+'>

div = <Arithmetic.div: '/'>

mul = <Arithmetic.mul: '*'>

sub = <Arithmetic.sub: '-'>

class pypika.enums.**Boolean**(**args*, ***kwds*)
Bases: *pypika.enums.Comparator*

and_ = <Boolean.and_: 'AND'>

or_ = <Boolean.or_: 'OR'>

xor_ = <Boolean.xor_: 'XOR'>

class pypika.enums.**Comparator**(**args*, ***kwds*)
Bases: aenum.Enum

class pypika.enums.**DatePart**(**args*, ***kwds*)
Bases: aenum.Enum

day = <DatePart.day: 'DAY'>

hour = <DatePart.hour: 'HOUR'>

microsecond = <DatePart.microsecond: 'MICROSECOND'>

minute = <DatePart.minute: 'MINUTE'>

month = <DatePart.month: 'MONTH'>

quarter = <DatePart.quarter: 'QUARTER'>

second = <DatePart.second: 'SECOND'>

week = <DatePart.week: 'WEEK'>

year = <DatePart.year: 'YEAR'>

class pypika.enums.**Dialects**(**args*, ***kwds*)
Bases: aenum.Enum

MSSQL = <Dialects.MSSQL: 'mssql'>

MYSQL = <Dialects.MYSQL: 'mysql'>

ORACLE = <Dialects.ORACLE: 'oracle'>

POSTGRESQL = <Dialects.POSTGRESQL: 'postgressql'>

REDSHIFT = <Dialects.REDSHIFT: 'redshift'>

VERTICA = <Dialects.VERTICA: 'vertica'>

class pypika.enums.**Equality**(**args*, ***kwds*)
Bases: *pypika.enums.Comparator*

eq = <Equality.eq: '='>

gt = <Equality.gt: '>'>

gte = <Equality.gte: '>='>

lt = <Equality.lt: '<'>

lte = <Equality.lte: '<='>

ne = <Equality.ne: '<>'>

class pypika.enums.**JoinType**(**args*, ***kwds*)
Bases: aenum.Enum

inner = <JoinType.inner: ‘’>

left = <JoinType.left: ‘LEFT’>

outer = <JoinType.outer: ‘OUTER’>

right = <JoinType.right: ‘RIGHT’>

class pypika.enums.**Matching**(**args*, ***kwds*)
Bases: *pypika.enums.Comparator*

bin_regex = <Matching.bin_regex: ‘ REGEX BINARY ‘>

like = <Matching.like: ‘ LIKE ‘>

regex = <Matching.regex: ‘ REGEX ‘>

class pypika.enums.**Order**(**args*, ***kwds*)
Bases: aenum.Enum

asc = <Order.asc: ‘ASC’>

desc = <Order.desc: ‘DESC’>

class pypika.enums.**SqlTypes**(**args*, ***kwds*)
Bases: aenum.Enum

DATE = <SqlTypes.DATE: ‘DATE’>

SIGNED = <SqlTypes.SIGNED: ‘SIGNED’>

TIMESTAMP = <SqlTypes.TIMESTAMP: ‘TIMESTAMP’>

UNSIGNED = <SqlTypes.UNSIGNED: ‘UNSIGNED’>

VARCHAR = <SqlTypes.VARCHAR: ‘VARCHAR’>

utf8 = <SqlTypes.utf8: ‘utf8’>

class pypika.enums.**UnionType**(**args*, ***kwds*)
Bases: aenum.Enum

all = <UnionType.all: ‘ ALL’>

distinct = <UnionType.distinct: ‘’>

pypika.functions module

Package for SQL functions wrappers

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_value*, *alias=None*)
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.terms.AggregateFunction*

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

get_function_sql(***kwargs*)

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

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

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

get_special_params_sql(***kwargs*)

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

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

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

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

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

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

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

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

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.**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*

pypika.queries module

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

get_sql(***kwargs*)

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

get_sql(***kwargs*)

validate(*_from*, *_joins*)

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

get_sql(***kwargs*)

validate(*_from*, *_joins*)

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

on(*criterion*)

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 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

class pypika.queries.**QueryBuilder**(*quote_char="'", dialect=None, wrap_union_queries=True*)
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.

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

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

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

do_join(*join*)

fields()

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

field(*name*)

star

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

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

pypika.queries.**make_tables**(**names*, ***kwargs*)

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()

for_(**args*, ***kwargs*)

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

is_aggregate

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

tables_

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

fields()

for_(**args*, ***kwargs*)

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

tables_

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

fields()

for_(**args*, ***kwargs*)

get_sql(***kwargs*)

tables_

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

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

fields()

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

is_aggregate

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(***kwargs*)

negate()

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

fields()

get_sql()

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

for_(**args*, ***kwargs*)

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

tables_

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

fields()

for_(**args*, ***kwargs*)

get_function_sql(***kwargs*)

get_special_params_sql(***kwargs*)

get_sql(*with_alias=False*, *with_namespace=False*, *quote_char=None*, ***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.**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']

trim_pattern = <_sre.SRE_Pattern object>

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

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

class pypika.terms.**Not**(*term*)
Bases: object

fields()

get_sql(***kwargs*)

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

fields()

for_(**args*, ***kwargs*)

get_sql(***kwargs*)

tables_

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

fields()

get_sql(***kwargs*)

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

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*)

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

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

between(*lower*, *upper*)

bin_regex(*pattern*)

eq(*other*)

fields()

get_sql()

gt(*other*)

gte(*other*)

is_aggregate = False

isin(*arg*)

isnull()

like(*expr*)

lt(*other*)

lte(*other*)

ne(*other*)

negate()

notin(*arg*)

notnull()

regex(*pattern*)

tables_

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

fields()

get_sql(***kwargs*)

class pypika.terms.**ValueWrapper**(*value*)
Bases: *pypika.terms.Term*

fields()

get_sql(***kwargs*)

is_aggregate = None

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

class Edge(*value=None*)

get_frame_sql()

get_partition_sql(***kwargs*)

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

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

pypika.utils module

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

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

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

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

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

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

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

pypika.utils.**alias_sql**(*sql*, *alias*, *quote_char=None*)

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.**ignoredeepcopy**(*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.

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

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y