Pony ORM

Release 0.7.1

May 22, 2017

Contents

CHAPTER 1

What is Pony ORM?

Pony is an advanced object-relational mapper. An ORM allows developers to work with the content of a database in the form of objects. A relational database contains rows that are stored in tables. However, when writing a program in a high level object-oriented language, it is much more convenient when the data retrieved from the database can be accessed in the form of objects. Pony ORM is a library for Python language that allows you to conveniently work with objects that are stored as rows in a relational database.

There are other popular mappers implemented in Python such as Django and SQLAlchemy, but we believe that Pony has some distinct advantages:

- An exceptionally convenient syntax for writing queries
- Automatic query optimization
- An elegant solution for the N+1 problem
- The online database schema editor

In comparison to Django, Pony provides:

- The IdentityMap pattern
- Automatic transaction management
- Automatic caching of queries and objects
- Full support of composite keys
- The ability to easily write queries using LEFT JOIN, HAVING and other features of SQL

One interesting feature of Pony is that it allows interacting with the database in pure Python using the generator expressions or lambda functions, which are then translated into SQL. Such queries may easily be written by a developer familiar with Python, even without being a database expert. Here is an example of a query using the generator expression syntax:

```
select(c for c in Customer if sum(c.orders.total_price) > 1000)
```

Here is the same query written using the lambda function:

```
Customer.select(lambda c: sum(c.orders.total_price) > 1000)
```

In this query, we retrieve all customers with total amount of purchases exceeding 1000. The query to the database is described in the form of Python generator expression and passed to the *select()* function as an argument. Pony doesn't execute this generator, but translates it into SQL and then sends to the database. Using such approach any developer can write database queries without being an expert in SQL.

The Customer is an entity class that is initially described when the application is created, and linked to a table in the database.

Not every object-relational mapper offers such a convenient query syntax. In addition to ease of use, Pony ensures efficient work with data. Queries are translated into SQL that is executed quickly and efficiently. Depending on the DBMS, the syntax of the generating SQL may vary in order to use features of the chosen database. The query code written in Python will look the same regardless of the DBMS, which ensures the application's portability.

With Pony any developer can write complex and effective queries, even without being an expert in SQL. At the same time, Pony does not "fight" with SQL – if a developer needs to write a query in raw SQL, for example to call up a stored procedure, he or she can easily do this with Pony. The main goal of Pony ORM is to simplify the process of development of web applications.

Starting with the version 0.7, Pony ORM is released under the Apache License, Version 2.0.

PonyORM community

If you have any questions, please post them on Stack Overflow. Meet the PonyORM team, chat with the community members, and get your questions answered on our community Telegram group. Join our newsletter at ponyorm.com. Reach us on Twitter or contact the Pony ORM team by e-mail: team (at) ponyorm.com.

CHAPTER 2

Getting Started with Pony

Installing

To install Pony, type the following command into the command prompt:

```
pip install pony
```

Pony can be installed on Python 2.7 or Python 3. If you are going to work with SQLite database, you don't need to install anything else. If you wish to use another database, you need to have the corresponding database driver installed:

- PostgreSQL: psycopg or psycopg2cffi
- MySQL: MySQL-python or PyMySQL
- Oracle: cx_Oracle

To make sure Pony has been successfully installed, launch a Python interpreter in interactive mode and type:

```
>>> from pony.orm import *
```

This imports the entire (and not very large) set of classes and functions necessary for working with Pony. Eventually you can choose what to import, but we recommend using `import *` at first.

If you don't want to import everything into global namespace, you can import the `orm` package only:

```
>>> from pony import orm
```

In this case you don't load all Pony's functions into the global namespace, but it will require you to use `orm` as a prefix to any Pony's function and decorator.

The best way to become familiar with Pony is to play around with it in interactive mode. Let's create a sample database containing the entity class `Person`, add three objects to it, and write a query.

Creating the database object

Entities in Pony are connected to a database. This is why we need to create the database object first. In the Python interpreter, type:

```
>>> db = Database()
```

Defining entities

Now, let's create two entities – Person and Car. The entity Person has two attributes – name and age, and Car has attributes make and model. In the Python interpreter, type the following code:

```
>>> class Person(db.Entity):
...     name = Required(str)
...     age = Required(int)
...     cars = Set('Car')
...
>>> class Car(db.Entity):
...     make = Required(str)
...     model = Required(str)
...     owner = Required(Person)
...
>>>
```

The classes that we have created are derived from the *Database.Entity* attribute of the *Database* object. It means that they are not ordinary classes, but entities. The entity instances are stored in the database, which is bound to the db variable. With Pony you can work with several databases at the same time, but each entity belongs to one specific database.

Inside the entity Person we have created three attributes – name, age and cars. The name and age are mandatory attributes. In other words, they these attributes cannot have the None value. The name is a string attribute, while age is numeric.

The cars attribute is declared as *Set* and has the Car type. This means that this is a relationship. It can keep a collection of instances of the Car entity. "Car" is specified as a string here because we didn't declare the entity Car by that moment yet.

The Car entity has three mandatory attributes: make and model are strings, and the owner attribute is the other side of the one-to-many relationship. Relationships in Pony are always defined by two attributes which represent both sides of a relationship.

If we need to create a many-to-many relationship between two entities, we should declare two *Set* attributes at both ends. Pony creates the intermediate database table automatically.

The str type is used for representing an unicode string in Python 3. Python 2 has two types for strings - str and unicode. Starting with the Pony Release 0.6, you can use either str or unicode for string attributes, both of them mean an unicode string. We recommend using the str type for string attributes, because it looks more natural in Python 3.

If you need to check an entity definition in the interactive mode, you can use the *show()* function. Pass the entity class or the entity instance to this function for printing out the definition:

```
>>> show(Person)
class Person(Entity):
    id = PrimaryKey(int, auto=True)
    name = Required(str)
```

```
    age = Required(int)
    cars = Set(Car)
```

You may notice that the entity got one extra attribute named `id`. Why did that happen?

Each entity must contain a primary key, which allows distinguishing one entity from the other. Since we have not set the primary key attribute manually, it was created automatically. If the primary key is created automatically, it is named as `id` and has a numeric format. If the primary key attribute is created manually, you can specify the name and type of your choice. Pony also supports composite primary keys.

When the primary key is created automatically, it always has the option `auto` set to `True`. It means that the value for this attribute will be assigned automatically using the database's incremental counter or a database sequence.

Database binding

The database object has the *Database.bind()* method. It is used for attaching declared entities to a specific database. If you want to play with Pony in the interactive mode, you can use the SQLite database created in memory:

```
>>> db.bind('sqlite', ':memory:')
```

The first parameter specifies the database type that we want to work with. Currently Pony supports 4 database types: `'sqlite'`, `'mysql'`, `'postgresql'` and `'oracle'`. The subsequent parameters are specific to each database. They are the same ones that you would use if you were connecting to the database through the DB-API module.

For SQLite, either the database filename or the string ':memory:' must be specified as the parameter, depending on where the database is being created. If the database is created in-memory, it will be deleted once the interactive session in Python is over. In order to work with the database stored in a file, you can replace the previous line with the following:

```
>>> db.bind('sqlite', 'database.sqlite', create_db=True)
```

In this case, if the database file does not exist, it will be created. In our example, we can use a database created in-memory.

If you're using another database, you need to have the specific database adapter installed. For PostgreSQL Pony uses psycopg2. For MySQL either MySQLdb or pymysql adapter. For Oracle Pony uses the cx_Oracle adapter.

Here is how you can get connected to the databases:

```
# SQLite
db.bind('sqlite', ':memory:')
# or
db.bind('sqlite', 'database_file.sqlite', create_db=True)

# PostgreSQL
db.bind('postgres', user='', password='', host='', database='')

# MySQL
db.bind('mysql', host='', user='', passwd='', db='')

# Oracle
db.bind('oracle', 'user/password@dsn')
```

Mapping entities to database tables

Now we need to create database tables where we will persist our data. For this purpose, we need to call the *generate_mapping()* method on the *Database* object:

```
>>> db.generate_mapping(create_tables=True)
```

The parameter `create_tables=True` indicates that, if the tables do not already exist, then they will be created using the `CREATE TABLE` command.

All entities connected to the database must be defined before calling *generate_mapping()* method.

Using the debug mode

Using the *sql_debug()* function, you can see the SQL commands that Pony sends to the database. In order to turn the debug mode on, type the following:

```
>>> sql_debug(True)
```

If this command is executed before calling the *generate_mapping()* method, then during the creation of the tables, you will see the SQL code used to generate them.

Creating entity instances

Now, let's create five objects that describe three persons and two cars, and save this information in the database:

```
>>> p1 = Person(name='John', age=20)
>>> p2 = Person(name='Mary', age=22)
>>> p3 = Person(name='Bob', age=30)
>>> c1 = Car(make='Toyota', model='Prius', owner=p2)
>>> c2 = Car(make='Ford', model='Explorer', owner=p3)
>>> commit()
```

Pony does not save objects in the database immediately. These objects will be saved only after the *commit()* function is called. If the debug mode is turned on, then during the *commit()*, you will see five `INSERT` commands sent to the database.

db_session

The code which interacts with the database has to be placed within a database session. When you work with Python's interactive shell you don't need to worry about the database session, because it is maintained by Pony automatically. But when you use Pony in your application, all database interactions should be done within a database session. In order to do that you need to wrap the functions that work with the database with the *db_session()* decorator:

```
@db_session
def print_person_name(person_id):
    p = Person[person_id]
    print p.name
    # database session cache will be cleared automatically
    # database connection will be returned to the pool
```

```
@db_session
def add_car(person_id, make, model):
    Car(make=make, model=model, owner=Person[person_id])
    # commit() will be done automatically
    # database session cache will be cleared automatically
    # database connection will be returned to the pool
```

The *db_session()* decorator performs the following actions on exiting function:

- Performs rollback of transaction if the function raises an exception
- Commits transaction if data was changed and no exceptions occurred
- Returns the database connection to the connection pool
- Clears the database session cache

Even if a function just reads data and does not make any changes, it should use the *db_session()* in order to return the connection to the connection pool.

The entity instances are valid only within the *db_session()*. If you need to render an HTML template using those objects, you should do this within the *db_session()*.

Another option for working with the database is using the *db_session()* as the context manager instead of the decorator:

```
with db_session:
    p = Person(name='Kate', age=33)
    Car(make='Audi', model='R8', owner=p)
    # commit() will be done automatically
    # database session cache will be cleared automatically
    # database connection will be returned to the pool
```

Writing queries

Now that we have the database with five objects saved in it, we can try some queries. For example, this is the query which returns a list of persons who are older than twenty years old:

```
>>> select(p for p in Person if p.age > 20)
<pony.orm.core.Query at 0x105e74d10>
```

The *select()* function translates the Python generator into a SQL query and returns an instance of the *Query* class. This SQL query will be sent to the database once we start iterating over the query. One of the ways to get the list of objects is to apply the slice operator `[:]` to it:

```
>>> select(p for p in Person if p.age > 20)[:]

SELECT "p"."id", "p"."name", "p"."age"
FROM "Person" "p"
WHERE "p"."age" > 20

[Person[2], Person[3]]
```

As the result you can see the text of the SQL query which was sent to the database and the list of extracted objects. When we print out the query result, the entity instance is represented by the entity name and its primary key written in square brackets, e.g. `Person[2]`.

For ordering the resulting list you can use the *Query.order_by()* method. If you need only a portion of the result set, you can use the slice operator, the exact same way as you would do that on a Python list. For example, if you want to sort all people by their name and extract the first two objects, you do it this way:

```
>>> select(p for p in Person).order_by(Person.name)[:2]

SELECT "p"."id", "p"."name", "p"."age"
FROM "Person" "p"
ORDER BY "p"."name"
LIMIT 2

[Person[3], Person[1]]
```

Sometimes, when working in the interactive mode, you might want to see the values of all object attributes. For this purpose, you can use the *Query.show()* method:

```
>>> select(p for p in Person).order_by(Person.name)[:2].show()

SELECT "p"."id", "p"."name", "p"."age"
FROM "Person" "p"
ORDER BY "p"."name"
LIMIT 2

id|name|age
--+----+---
3 |Bob |30
1 |John|20
```

The *Query.show()* method doesn't display "to-many" attributes because it would require additional query to the database and could be bulky. That is why you can see no information about the related cars above. But if an instance has a "to-one" relationship, then it will be displayed:

```
>>> Car.select().show()
id|make  |model   |owner
--+------+--------+---------
1 |Toyota|Prius   |Person[2]
2 |Ford  |Explorer|Person[3]
```

If you don't want to get a list of objects, but need to iterate over the resulting sequence, you can use the `for` loop without using the slice operator:

```
>>> persons = select(p for p in Person if 'o' in p.name)
>>> for p in persons:
...     print p.name, p.age
...
SELECT "p"."id", "p"."name", "p"."age"
FROM "Person" "p"
WHERE "p"."name" LIKE '%o%'

John 20
Bob 30
```

In the example above we get all Person objects with the name attribute containing the letter 'o' and display the person's name and age.

A query does not necessarily have to return entity objects. For example, you can get a list, consisting of the object attribute:

```
>>> select(p.name for p in Person if p.age != 30)[:]

SELECT DISTINCT "p"."name"
FROM "Person" "p"
WHERE "p"."age" <> 30

[u'John', u'Mary']
```

Or a list of tuples:

```
>>> select((p, count(p.cars)) for p in Person)[:]

SELECT "p"."id", COUNT(DISTINCT "car-1"."id")
FROM "Person" "p"
  LEFT JOIN "Car" "car-1"
    ON "p"."id" = "car-1"."owner"
GROUP BY "p"."id"

[(Person[1], 0), (Person[2], 1), (Person[3], 1)]
```

In the example above we get a list of tuples consisting of a `Person` object and the number of cars they own.

With Pony you can also run aggregate queries. Here is an example of a query which returns the maximum age of a person:

```
>>> print max(p.age for p in Person)
SELECT MAX("p"."age")
FROM "Person" "p"

30
```

In the following parts of this manual you will see how you can write more complex queries.

Getting objects

To get an object by its primary key you need to specify the primary key value in the square brackets:

```
>>> p1 = Person[1]
>>> print p1.name
John
```

You may notice that no query was sent to the database. That happened because this object is already present in the database session cache. Caching reduces the number of requests that need to be sent to the database.

For retrieving the objects by other attributes, you can use the *`Entity.get()`* method:

```
>>> mary = Person.get(name='Mary')

SELECT "id", "name", "age"
FROM "Person"
WHERE "name" = ?
[u'Mary']

>>> print mary.age
22
```

In this case, even though the object had already been loaded to the cache, the query still had to be sent to the database because the `name` attribute is not a unique key. The database session cache will only be used if we lookup an object by its primary or unique key.

You can pass an entity instance to the *show()* function in order to display the entity class and attribute values:

```
>>> show(mary)
instance of Person
id|name|age
--+----+---
2 |Mary|22
```

Updating an object

```
>>> mary.age += 1
>>> commit()
```

Pony keeps track of all changed attributes. When the *commit()* function is executed, all objects that were updated during the current transaction will be saved in the database. Pony saves only those attributes, that were changed during the database session.

Writing raw SQL queries

If you need to select entities by a raw SQL query, you can do it this way:

```
>>> x = 25
>>> Person.select_by_sql('SELECT * FROM Person p WHERE p.age < $x')

SELECT * FROM Person p WHERE p.age < ?
[25]

[Person[1], Person[2]]
```

If you want to work with the database directly, avoiding entities, you can use the *Database.select()* method:

```
>>> x = 20
>>> db.select('name FROM Person WHERE age > $x')
SELECT name FROM Person WHERE age > ?
[20]

[u'Mary', u'Bob']
```

Pony examples

Instead of creating models manually, you can check the examples from the Pony distribution package:

```
>>> from pony.orm.examples.estore import *
```

Here you can see the database diagram for this example: https://editor.ponyorm.com/user/pony/eStore.

During the first import, there will be created the SQLite database with all the necessary tables. In order to fill it in with the data, you need to call the following function:

```
>>> populate_database()
```

This function will create objects and place them in the database.

After the objects have been created, you can try some queries. For example, here is how you can display the country where we have most of the customers:

```
>>> select((customer.country, count(customer))
...         for customer in Customer).order_by(-2).first()

SELECT "customer"."country", COUNT(DISTINCT "customer"."id")
FROM "Customer" "customer"
GROUP BY "customer"."country"
ORDER BY 2 DESC
LIMIT 1
```

In this example, we are grouping objects by the country, sorting them by the second column (the number of customers) in the reverse order, and then extracting the first row.

You can find more query examples in the `test_queries()` function in the pony.orm.examples.estore module.

CHAPTER 3

Connecting to the Database

Before you can start working with entities you have to create the `Database` object. The entities, that you declare in your Python code, will be mapped to the database through this object.

Mapping entities to the database can be divided into four steps:

- Creating the *Database* object
- Defining entities which are related to this *Database* object
- Binding the *Database* object to a specific database
- Mapping entities to the database tables

Now we'll describe the main workflow of working with the *Database* object and its methods. When you'll need more details on this, you can find them in the *API Reference*.

Creating the Database object

At this step we simply create an instance of the *Database* class:

```
db = Database()
```

The *Database* class instance has an attribute *Entity* which represents a base class to be used for entities declaration.

Defining entities which are related to the Database object

Entities should inherit from the base class of the *Database* object:

```
class MyEntity(db.Entity):
    attr1 = Required(str)
```

We'll talk about entities definition in detail in the next chapter. Now, let's see the next step in mapping entities to a database.

Binding the database object to a specific database

Before we can map entities to the database, we need to connect to establish connection to it. It can be done using the *bind()* method:

```
db.bind('postgres', user='', password='', host='', database='')
```

The first parameter of this method is the name of the database provider. The database provider is a module which resides in the `pony.orm.dbproviders` package and which knows how to work with a particular database. After the database provider name you should specify parameters which will be passed to the `connect()` method of the corresponding DBAPI driver.

Currently Pony can work with four database systems: SQLite, PostgreSQL, MySQL and Oracle, with the corresponding Pony provider names: `'sqlite'`, `'postgres'`, `'mysql'` and `'oracle'`. Pony can easily be extended to incorporate additional database providers.

When you just start working with Pony, you can use the SQLite database. This database is included into Python distribution and you don't need to install anything separately. Using SQLite you can create the database either in a file or in memory. For creating the database in the file use the following command:

```
db.bind('sqlite', 'database.sqlite', create_db=True)
```

When `create_db=True`, Pony will create the database file if it doesn't exist. If it already exists, Pony will use it.

For in-memory database use this:

```
db.bind('sqlite', ':memory:')
```

There is no need in the parameter `create_db` when creating an in-memory database. This is a convenient way to create a SQLite database when playing with Pony in the interactive shell, but you should remember, that the entire in-memory database will be lost on program exit.

Here are the examples of binding to other databases:

```
db.bind('sqlite', ':memory:')
db.bind('sqlite', 'filename', create_db=True)
db.bind('mysql', host='', user='', passwd='', db='')
db.bind('oracle', 'user/password@dsn')
```

You can find more details on working with each database in the API Reference:

- *SQLite*
- *PostgreSQL*
- *MySQL*
- *Oracle*

Mapping entities to the database tables

After the *Database* object is created, entities are defined, and a database is bound, the next step is to map entities to the database tables using the *generate_mapping()* method:

```
db.generate_mapping(create_tables=True)
```

This method creates tables, foreign key references and indexes if they don't exist. After entities are mapped, you can start working with them in your Python code - select, create, modify objects and save them in the database.

Methods and attributes of the Database object

The *Database* object has a set of methods, which you can exampne in the *API Reference*.

Using Database object for raw SQL queries

Typically you will work with entities and let Pony interact with the database, but Pony also allows you to work with the database using SQL, or even combine both ways. Of course you can work with the database directly using the DBAPI interface, but using the Database object gives you the following advantages:

- Automatic transaction management using the *db_session()* decorator or context manager. All data will be stored to the database after the transaction is finished, or rolled back if an exception happened.
- Connection pool. There is no need to keep track of database connections. You have the connection when you need it and when you have finished your transaction the connection will be returned to the pool.
- Unified database exceptions. Each DBAPI module defines its own exceptions. Pony allows you to work with the same set of exceptions when working with any database. This helps you to create applications which can be ported from one database to another.
- Unified way of passing parameters to SQL queries with the protection from injection attacks. Different database drivers use different paramstyles - the DBAPI specification offers 5 different ways of passing parameters to SQL queries. Using the *Database* object you can use one way of passing parameters for all databases and eliminate the risk of SQL injection.
- Automatic unpacking of single column results when using *get()* or *select()* methods of the *Database* object. If the *select()* method returns just one column, Pony returns a list of values, not a list of tuples each of which has just one item, as it does DBAPI. If the *get()* method returns a single column it returns just value, not a tuple consisting of one item. It's just convenient.
- When the methods *select()* or *get()* return more than one column, Pony uses smart tuples which allow accessing items as tuple attributes using column names, not just tuple indices.

In other words the *Database* object helps you save time completing routine tasks and provides convenience and uniformity.

Using parameters in raw SQL queries

With Pony you can easily pass parameters into SQL queries. In order to specify a parameter you need to put the *$* sign before the variable name:

```
x = "John"
data = db.select("select * from Person where name = $x")
```

When Pony encounters such a parameter within the SQL query it gets the variable value from the current frame (from globals and locals) or from the dictionary which is passed as the second parameter. In the example above Pony will try to get the value for `$x` from the variable `x` and will pass this value as a parameter to the SQL query which eliminates the risk of SQL injection. Below you can see how to pass a dictionary with the parameters:

```
data = db.select("select * from Person where name = $x", {"x" : "Susan"})
```

This method of passing parameters to the SQL queries is very flexible and allows using not only single variables, but any Python expression. In order to specify an expression you need to put it in parentheses after the $ sign:

```
data = db.select("select * from Person where name = $(x.lower()) and age > $(y + 2)")
```

All the parameters can be passed into the query using the Pony unified way, independently of the DBAPI provider, using the `$` sign. In the example above we pass `name` and `age` parameters into the query.

It is possible to have a Python expressions inside the query text, for example:

```
x = 10
a = 20
b = 30
db.execute("SELECT * FROM Table1 WHERE column1 = $x and column2 = $(a + b)")
```

If you need to use the $ sign as a string literal inside the query, you need to escape it using another $ (put two $ signs in succession: $$).

Database statistics

The `Database` object keeps statistics on executed queries. You can check which queries were executed more often and how long it took to execute them as well as some other parameters. Check the *`QueryStat`* class in the API Reference for more details.

CHAPTER 4

Declaring Entities

Entities are Python classes which store an object's state in the database. Each instance of an entity corresponds to a row in the database table. Often entities represent objects from the real world (e.g. Customer, Product).

Before creating entity instances you need to map entities to the database tables. Pony can map entities to existing tables or create new tables. After the mapping is generated you can query the database and create new instances of entities.

Pony provides an entity-relationship diagram editor which can be used for creating Python entity declarations.

Declaring an entity

Each entity belongs to a database. That is why before defining entities you need to create an object of the `Database` class:

```
from pony.orm import *

db = Database()

class MyEntity(db.Entity):
    attr1 = Required(str)
```

The Pony's Database object has the `Entity` attribute which is used as a base class for all the entities stored in this database. Each new entity that is defined must inherit from this `Entity` class.

Entity attributes

Entity attributes are specified as class attributes inside the entity class using the syntax `attr_name = kind(type, options)`:

```
class Customer(db.Entity):
    name = Required(str)
    email = Required(str, unique=True)
```

In the parentheses, after the attribute type, you can specify attribute options.

Each attribute can be one of the following kinds:

- Required
- Optional
- PrimaryKey
- Set

Required and Optional

Usually most entity attributes are of `Required` or `Optional` kind. If an attribute is defined as `Required` then it must have a value at all times, while `Optional` attributes can be empty.

If you need the value of an attribute to be unique then you can set the attribute option `unique=True`.

PrimaryKey

`PrimaryKey` defines an attribute which is used as a primary key in the database table. Each entity should always have a primary key. If the primary key is not specified explicitly, Pony will create it implicitly. Let's consider the following example:

```
class Product(db.Entity):
    name = Required(str, unique=True)
    price = Required(Decimal)
    description = Optional(str)
```

The entity definition above will be equal to the following:

```
class Product(db.Entity):
    id = PrimaryKey(int, auto=True)
    name = Required(str, unique=True)
    price = Required(Decimal)
    description = Optional(str)
```

The primary key attribute which Pony adds automatically always will have the name `id` and `int` type. The option `auto=True` means that the value for this attribute will be assigned automatically using the database's incremental counter or a sequence.

If you specify the primary key attribute yourself, it can have any name and type. For example, we can define the entity `Customer` and have customer's email as the primary key:

```
class Customer(db.Entity):
   email = PrimaryKey(str)
   name = Required(str)
```

Set

A `Set` attribute represents a collection. Also we call it a relationship, because such attribute relates to an entity. You need to specify an entity as the type for the `Set` attribute. This is the way to define one side for the to-many relationships. As of now, Pony doesn't allow the use of `Set` with primitive types. We plan to add this feature later.

We will talk in more detail about this attribute type in the *Entity relationships* chapter.

Composite keys

Pony fully supports composite keys. In order to declare a composite primary key you need to specify all the parts of the key as `Required` and then combine them into a composite primary key:

```
class Example(db.Entity):
    a = Required(int)
    b = Required(str)
    PrimaryKey(a, b)
```

Here `PrimaryKey(a, b)` doesn't create an attribute, but combines the attributes specified in the parenthesis into a composite primary key. Each entity can have only one primary key.

In order to declare a secondary composite key you need to declare attributes as usual and then combine them using the `composite_key` directive:

```
class Example(db.Entity):
    a = Required(str)
    b = Optional(int)
    composite_key(a, b)
```

In the database `composite_key(a, b)` will be represented as the `UNIQUE ("a", "b")` constraint.

If have just one attribute, which represents a unique key, you can create such a key by specifying `unique=True` by an attribute:

```
class Product(db.Entity):
    name = Required(str, unique=True)
```

Composite indexes

Using the `composite_index()` directive you can create a composite index for speeding up data retrieval. It can combine two or more attributes:

```
class Example(db.Entity):
    a = Required(str)
    b = Optional(int)
    composite_index(a, b)
```

You can use the attribute or the attribute name:

```
class Example(db.Entity):
    a = Required(str)
    b = Optional(int)
    composite_index(a, 'b')
```

If you want to create a non-unique index for just one column, you can specify the `index` option of an attribute.

The composite index can include a discriminator attribute used for inheritance.

Attribute data types

Pony supports the following attribute types:

- str
- unicode
- int
- float
- Decimal
- datetime
- date
- time
- timedelta
- bool
- buffer - used for binary data in Python 2 and 3
- bytes - used for binary data in Python 3
- LongStr - used for large strings
- LongUnicode - used for large strings
- UUID
- Json - used for mapping to native database JSON type

See the *Attribute types* part of the API Reference for more information.

Attribute options

You can specify additional options during attribute definitions using positional and keyword arguments. See the *Attribute options* part of the API Reference for more information.

Entity inheritance

Entity inheritance in Pony is similar to inheritance for regular Python classes. Let's consider an example of a data diagram where entities `Student` and `Professor` inherit from the entity `Person`:

```
class Person(db.Entity):
    name = Required(str)

class Student(Person):
    gpa = Optional(Decimal)
    mentor = Optional("Professor")

class Professor(Person):
    degree = Required(str)
    students = Set("Student")
```

All attributes and relationships of the base entity `Person` are inherited by all descendants.

In some mappers (e.g. Django) a query on a base entity doesn't return the right class: for derived entities the query returns just a base part of each instance. With Pony you always get the correct entity instances:

```
for p in Person.select():
    if isinstance(p, Professor):
        print p.name, p.degree
    elif isinstance(p, Student):
        print p.name, p.gpa
    else:  # somebody else
        print p.name
```

In order to create the correct entity instance Pony uses a discriminator column. By default this is a string column and Pony uses it to store the entity class name:

```
classtype = Discriminator(str)
```

By default Pony implicitly creates the `classtype` attribute for each entity class which takes part in inheritance. You can use your own discriminator column name and type. If you change the type of the discriminator column, then you have to specify the `_discrimintator_` value for each entity.

Let's consider the example above and use `cls_id` as the name for our discriminator column of `int` type:

```
class Person(db.Entity):
    cls_id = Discriminator(int)
    _discriminator_ = 1
    ...

class Student(Person):
    _discriminator_ = 2
    ...

class Professor(Person):
    _discriminator_ = 3
    ...
```

Multiple inheritance

Pony also supports multiple inheritance. If you use multiple inheritance then all the parent classes of the newly defined class should inherit from the same base class (a "diamond-like" hierarchy).

Let's consider an example where a student can have a role of a teaching assistant. For this purpose we'll introduce the entity `Teacher` and derive `Professor` and `TeachingAssistant` from it. The entity `TeachingAssistant` inherits from both the `Student` class and the `Teacher` class:

```
class Person(db.Entity):
    name = Required(str)

class Student(Person):
    ...

class Teacher(Person):
    ...

class Professor(Teacher):
    ...
```

```
class TeachingAssistant(Student, Teacher):
    ...
```

The `TeachingAssistant` objects are instances of both `Teacher` and `Student` entities and inherit all their attributes. Multiple inheritance is possible here because both `Teacher` and `Student` have the same base class `Person`.

Inheritance is a very powerful tool, but it should be used wisely. Often the data diagram is much simpler if it has limited usage of inheritance.

Representing inheritance in the database

There are three ways to implement inheritance in the database:

1. Single Table Inheritance: all entities in the hierarchy are mapped to a single database table.
2. Class Table Inheritance: each entity in the hierarchy is mapped to a separate table, but each table stores only the attributes which the entity doesn't inherit from its parents.
3. Concrete Table Inheritance: each entity in the hierarchy is mapped to a separate table and each table stores the attributes of the entity and all its ancestors.

The main problem of the third approach is that there is no single table where we can store the primary key and that is why this implementation is rarely used.

The second implementation is used often, this is how the inheritance is implemented in Django. The disadvantage of this approach is that the mapper has to join several tables together in order to retrieve data which can lead to the performance degradation.

Pony uses the first approach where all entities in the hierarchy are mapped to a single database table. This is the most efficient implementation because there is no need to join tables. This approach has its disadvantages too:

- Each table row has columns which are not used because they belong to other entities in the hierarchy. It is not a big problem because the blank columns keep `NULL` values and it doesn't use much space.
- The table can have large number of columns if there are a lot of entities in the hierarchy. Different databases have different limits for maximum columns per table, but usually that limit is pretty high.

Adding custom methods to entities

Besides data attributes, entities can have methods. The most straightforward way of adding methods to entities is defining those methods in the entity class. Let's say we would like to have a method of the Product entity which returns concatenated name and price. It can be done the following way:

```
class Product(db.Entity):
    name = Required(str, unique=True)
    price = Required(Decimal)

    def get_name_and_price(self):
        return "%s (%s)" % (self.name, self.price)
```

Another approach is using mixin classes. Instead of putting custom methods directly to the entity definition, you can define them in a separate mixin class and inherit entity class from that mixin:

```
class ProductMixin(object):
    def get_name_and_price(self):
        return "%s (%s)" % (self.name, self.price)

class Product(db.Entity, ProductMixin):
    name = Required(str, unique=True)
    price = Required(Decimal)
```

This approach can be beneficial if you are using our online ER diagram editor. The editor automatically generates entity definitions in accordance with the diagram. In this case, if you add some custom methods to the entity definition, these methods will be overwritten once you change your diagram and save newly generated entity definitions. Using mixins would allow you to separate entity definitions and mixin classes with methods into two different files. This way you can overwrite your entity definitions without losing your custom methods.

For our example above the separation can be done in the following way.

File mixins.py:

```
class ProductMixin(object):
    def get_name_and_price(self):
        return "%s (%s)" % (self.name, self.price)
```

File models.py:

```
from decimal import Decimal
from pony.orm import *
from mixins import *

class Product(db.Entity, ProductMixin):
    name = Required(str, unique=True)
    price = Required(Decimal)
```

Mapping customization

When Pony creates tables from entity definitions, it uses the name of entity as the table name and attribute names as the column names, but you can override this behavior.

The name of the table is not always equal to the name of an entity: in MySQL and PostgreSQL the default table name generated from the entity name will be converted to the lower case, in Oracle - to the upper case. You can always find the name of the entity table by reading the `_table_` attribute of an entity class.

If you need to set your own table name use the `_table_` class attribute:

```
class Person(db.Entity):
    _table_ = "person_table"
    name = Required(str)
```

If you need to set your own column name, use the option `column`:

```
class Person(db.Entity):
    _table_ = "person_table"
    name = Required(str, column="person_name")
```

For composite attributes use the option `columns` with the list of the column names specified:

```
class Course(db.Entity):
    name = Required(str)
    semester = Required(int)
    lectures = Set("Lecture")
    PrimaryKey(name, semester)

class Lecture(db.Entity):
    date = Required(datetime)
    course = Required(Course, columns=["name_of_course", "semester"])
```

In this example we override the column names for the composite attribute `Lecture.course`. By default Pony will generate the following column names: `"course_name"` and `"course_semester"`. Pony combines the entity name and the attribute name in order to make the column names easy to understand to the developer.

If you need to set the column names for the intermediate table for many-to-many relationship, you should specify the option `column` or `columns` for the `Set` attributes. Let's consider the following example:

```
class Student(db.Entity):
    name = Required(str)
    courses = Set("Course")

class Course(db.Entity):
    name = Required(str)
    semester = Required(int)
    students = Set(Student)
    PrimaryKey(name, semester)
```

By default, for storing many-to-many relationships between `Student` and `Course`, Pony will create an intermediate table `"Course_Student"` (it constructs the name of the intermediate table from the entity names in the alphabetical order). This table will have three columns: `"course_name"`, `"course_semester"` and `"student"` - two columns for the `Course`'s composite primary key and one column for the `Student`. Now let's say we want to name the intermediate table as `"Study_Plans"` which have the following columns: `"course"`, `"semester"` and `"student_id"`. This is how we can achieve this:

```
class Student(db.Entity):
    name = Required(str)
    courses = Set("Course", table="Study_Plans", columns=["course", "semester"]))

class Course(db.Entity):
    name = Required(str)
    semester = Required(int)
    students = Set(Student, column="student_id")
    PrimaryKey(name, semester)
```

You can find more examples of mapping customization in an example which comes with Pony ORM package

CHAPTER 5

JSON Data Type Support

Overview

Recently native JSON data type support has been added in all major database systems. JSON support introduces dynamic data structures commonly found in NoSQL databases. Usually they are used when working with highly varying data or when the exact data structure is hard to predict.

Pony allows working with JSON data stored in your database using Python syntax.

Declaring a JSON attribute

For declaring a JSON attribute with Pony you should use the `Json` type. This type can imported from `pony.orm` package:

```
from pony.orm import *

db = Database()


class Product(db.Entity):
    id = PrimaryKey(int, auto=True)
    name = Required(str)
    info = Required(Json)
    tags = Optional(Json)


db.bind('sqlite', ':memory:', create_db=True)
db.generate_mapping(create_tables=True)
```

The `info` attribute in the `Product` entity is declared as `Json`. This allows us to have different JSON structures for different product types, and make queries to this data.

Assigning value to the JSON attribute

Usually, a JSON structure contains a combination of dictionaries and lists containing simple types, such as numbers, strings and boolean. Let's create a couple of objects with a simple JSON structure:

```
p1 = Product(name='Samsung Galaxy S7 edge',
             info={
                 'display': {
                    'size': 5.5,
                 },
                 'battery': 3600,
                 '3.5mm jack': True,
                 'SD card slot': True
                 'colors': ['Black', 'Grey', 'Gold'],
             },
             tags=['Smartphone', 'Samsung'])

p2 = Product(name='iPhone 6s',
             info={
                 'display': {
                    'size': 4.7,
                    'resolution': [750, 1334],
                    'multi-touch': True
                 },
                 'battery': 1810,
                 '3.5mm jack': True,
                 'colors': ['Silver', 'Gold', 'Space Gray', 'Rose Gold']
             },
             tags=['Smartphone', 'Apple', 'Retina'])
```

In Python code a JSON structure is represented with the help of the standard Python dict and list. In our example, the `info` attribute is assigned with a dict. The `tags` attribute keeps a list of tags. These attributes will be serialized to JSON and stored in the database on commit.

Reading JSON attribute

You can read a JSON attribute as any other entity attribute:

```
>>> Product[1].info
{'battery': 3600, '3.5mm jack': True, 'colors': ['Black', 'Grey', 'Gold'],
'display': 5.5}
```

Once JSON attribute is extracted from the database, it is deserialized and represented as a combination of dicts and lists. You can use the standard Python dict and list API for working with the value:

```
>>> Product[1].info['colors']
['Black', 'Grey', 'Gold']

>>> Product[1].info['colors'][0]
'Black'

>>> 'Black' in Product[1].info['colors']
True
```

Modifying JSON attribute

For modifying the JSON attribute value, you use the standard Python list and dict API as well:

```
>>> Product[1].info['colors'].append('Silver')
>>> Product[1].info['colors']
['Black', 'Grey', 'Gold', 'Silver']
```

Now, on commit, the changes will be stored in the database. In order to track the changes made in the JSON structure, Pony uses its own dict and list implementations which inherit from the standard Python dict and list.

Below is a couple more examples of how you can modify the the JSON value.

```
p = Product[1]

# assigning a new value
p.info['display']['size'] = 4.7

# popping a dict value
display_size = p.info['display'].pop('size')

# removing a dict key using del
del p.info['display']

# adding a dict key
p.info['display']['resolution'] = [1440, 2560]

# removing a list item
del p.info['colors'][0]

# replacing a list item
p.info['colors'][1] = ['White']

# replacing a number of list items
p.info['colors'][1:] = ['White']
```

All of the actions above are regular Python operations with attributes, lists and dicts.

Querying JSON structures

Native JSON support in databases allows not only read and modify structured data, but also making queries. It is a very powerful feature - the queries use the same syntax and run in the same ACID transactional environment, in the same time offering NoSQL capabilities of a document store inside the relational database.

Pony allows selecting objects by filtering them by JSON sub-elements. To access JSON sub-element Pony constructs JSON path expression which then will be used inside a SQL query:

```
# products with display size greater than 5
Product.select(lambda p: p.info['display']['size'] > 5)
```

In order to specify values you can use parameters:

```
x = 2048
# products with width resolution greater or equal to x
Product.select(lambda p: p.info['display']['resolution'][0] >= x)
```

In MySQL, PostgreSQL and SQLite it is also possible to use parameters inside JSON path expression:

```
index = 0
Product.select(lambda p: p.info['display']['resolution'][index] < 2000)

key = 'display'
Product.select(lambda p: p.info[key]['resolution'][index] > 1000)
```

Note: Oracle does not support parameters inside JSON paths. With Oracle you can use constant keys only.

For JSON array you can calculate length:

```
# products with more than 2 tags
Product.select(lambda p: len(p.info['tags']) > 2)
```

Another query example is checking if a string key is a part of a JSON dict or array:

```
# products which have the resolution specified
Product.select(lambda p: 'resolution' in p.info['display'])

# products of black color
Product.select(lambda p: 'Black' in p.info['colors'])
```

When you compare JSON sub-element with `None`, it will be evaluated to `True` in the following cases:

- When the sub-element contains JSON `null` value
- When the sub-element does not exist

```
Product.select(lambda p: p.info['SD card slot'] is None)
```

You can test JSON sub-element for truth value:

```
# products with multi-touch displays
select(p for p in Product if p.info['display']['multi-touch'])
```

In Python, the following values are treated as false for conditionals: `None`, 0, `False`, empty string, empty dict and empty list. Pony keeps this behavior for conditions applied for JSON structures. Also, if the JSON path is not found, it will be evaluated to false.

In previous examples we used JSON structures in query conditions. But it is also possible to retrieve JSON structures or extract its parts as the query result:

```
select(p.info['display'] for p in Product)
```

When retrieving JSON structures this way, they will not be linked to entity instances. This means that modification of such JSON structures will not be saved to the database. Pony tracks JSON changes only when you select an object and modify its attributes.

MySQL and Oracle allows using wildcards in JSON path. Pony support wildcards by using special syntax:

- [...] means 'any dictionary element'
- [:] means 'any list item'

Here is a query example:

```
select(p.info['display'][...] for p in Product)
```

The result of such query will be an array of JSON sub-elements. With the current situation of JSON support in databases, the wildcards can be used only in the expression part of the generator expression.

JSON Support in Databases

For storing JSON in the database Pony uses the following types:

- SQLite - TEXT
- PostgreSQL - JSONB (binary JSON)
- MySQL - JSON (binary JSON, although it doesn't have 'B' in the name)
- Oracle - CLOB

Starting with the version 3.9 SQLite provides the JSON1 extension module. This extension improves performance when working with JSON queries, although Pony can work with JSON in SQLite even without this module.

CHAPTER 6

Entity Relationships

Entities can relate to each other. A relationship between two entities is defined by using two attributes which specify both ends of a relationship:

```
class Customer(db.Entity):
    orders = Set('Order')

class Order(db.Entity):
    customer = Required(Customer)
```

In the example above we have two relationship attributes: `orders` and `customer`. When we define the entity `Customer`, the entity `Order` is not defined yet. That is why we have to put quotes around `Order` in the `orders` attribute. Another option is to use lambda:

```
class Customer(db.Entity):
    orders = Set(lambda: Order)
```

This can be useful if you want your IDE to check the names of declared entities and highlight typos.

Some mappers (e.g. Django) require defining relationships on one side only. Pony requires defining relationships on both sides explicitly (as The Zen of Python reads: Explicit is better than implicit), which allows the user to see all relationships from the perspective of each entity.

All relationships are bidirectional. If you update one side of a relationship, the other side will be updated automatically. For example, if we create an instance of `Order` entity, the customer's set of orders will be updated to include this new order.

There are three types of relationships: one-to-one, one-to-many and many-to-many. A one-to-one relationship is rarely used, most relations between entities are one-to-many and many-to-many. If two entities have one-to-one relationship it often means that they can be combined into a single entity. If your data diagram has a lot of one-to-one relationships, then it may signal that you need to reconsider entity definitions.

One-to-many relationship

Here is an example of one-to-many relationship:

```
class Order(db.Entity):
    items = Set("OrderItem")

class OrderItem(db.Entity):
    order = Required(Order)
```

In the example above the instance of `OrderItem` cannot exist without an order. If we want to allow an instance of `OrderItem` to exist without being assigned to an order, we can define the `order` attribute as `Optional`:

```
class Order(db.Entity):
    items = Set("OrderItem")

class OrderItem(db.Entity):
    order = Optional(Order)
```

Many-to-many relationship

In order to create many-to-many relationship you need to define both ends of the relationship as `Set` attributes:

```
class Product(db.Entity):
    tags = Set("Tag")

class Tag(db.Entity):
    products = Set(Product)
```

In order to implement this relationship in the database, Pony will create an intermediate table. This is a well known solution which allows you to have many-to-many relationships in relational databases.

One-to-one relationship

In order to create a one-to-one relationship, the relationship attributes should be defined as `Optional-Required` or as `Optional-Optional`:

```
class Person(db.Entity):
    passport = Optional("Passport")

class Passport(db.Entity):
    person = Required("Person")
```

Defining both attributes as `Required` is not allowed because it doesn't make sense.

Self-references

An entity can relate to itself using a self-reference relationship. Such relationships can be of two types: symmetric and non-symmetric. A non-symmetric relationship is defined by two attributes which belong to the same entity.

The specifics of the symmetrical relationship is that the entity has just one relationship attribute specified, and this attribute defines both sides of the relationship. Such relationship can be either one-to-one or many-to-many. Here are examples of self-reference relationships:

```
class Person(db.Entity):
    name = Required(str)
    spouse = Optional("Person", reverse="spouse") # symmetric one-to-one
    friends = Set("Person", reverse="friends")    # symmetric many-to-many
    manager = Optional("Person", reverse="employees") # one side of non-symmetric
    employees = Set("Person", reverse="manager") # another side of non-symmetric
```

Multiple relationships between two entities

When two entities have more than one relationship between them, Pony requires the reverse attributes to be specified. This is needed in order to let Pony know which pair of attributes are related to each other. Let's consider the data diagram where a user can write tweets and also can favorite them:

```
class User(db.Entity):
    tweets = Set("Tweet", reverse="author")
    favorites = Set("Tweet", reverse="favorited")

class Tweet(db.Entity):
    author = Required(User, reverse="tweets")
    favorited = Set(User, reverse="favorites")
```

In the example above we have to specify the option `reverse`. If you will try to generate mapping for the entities definition without the `reverse` specified, you will get the exception `pony.orm.core.ERDiagramError: "Ambiguous reverse attribute for Tweet.author"`. That happens because in this case the attribute `author` can technically relate either to the attribute `tweets` or to `favorites` and Pony has no information on which one to use.

CHAPTER 7

Transactions and db_session

A database transaction is a logical unit of work, which can consist of one or several queries. Transactions are atomic, which means that when a transaction makes changes to the database, either all the changes succeed when the transaction is committed, or all the changes are undone when the transaction is rolled back.

Pony provides automatic transaction management using the database session.

Working with db_session

The code which interacts with the database has to be placed within a database session. The session sets the borders of a conversation with the database. Each application thread which works with the database establishes a separate database session and uses a separate instance of an Identity Map. This Identity Map works as a cache, helping to avoid a database query when you access an object by its primary or unique key and it is already stored in the Identity Map. In order to work with the database using the database session you can use the `@db_session()` decorator or *db_session()* context manager. When the session ends it does the following actions:

- Commits transaction if data was changed and no exceptions occurred otherwise it rolls back transaction.
- Returns the database connection to the connection pool.
- Clears the Identity Map cache.

If you forget to specify the *db_session()* where necessary, Pony will raise the exception `TransactionError: db_session is required when working with the database`.

Example of using the `@db_session()` decorator:

```
@db_session
def check_user(username):
    return User.exists(username=username)
```

Example of using the *db_session()* context manager:

```
def process_request():
    ...
```

```
    with db_session:
        u = User.get(username=username)
        ...
```

Note: When you work with Python's interactive shell you don't need to worry about the database session, because it is maintained by Pony automatically.

If you'll try to access instance's attributes which were not loaded from the database outside of the *db_session()* scope, you'll get the `DatabaseSessionIsOver` exception. For example:

```
DatabaseSessionIsOver: Cannot load attribute Customer[3].name: the database session␣
↪is over
```

This happens because by this moment the connection to the database is already returned to the connection pool, transaction is closed and we cannot send any queries to the database.

When Pony reads objects from the database it puts those objects to the Identity Map. Later, when you update an object's attributes, create or delete an object, the changes will be accumulated in the Identity Map first. The changes will be saved in the database on transaction commit or before calling the following methods: `select()`, `get()`, `exists()`, `execute()`.

db_session and the transaction scope

Usually you will have a single transaction within the *db_session()*. There is no explicit command for starting a transaction. A transaction begins with the first SQL query sent to the database. Before sending the first query, Pony gets a database connection from the connection pool. Any following SQL queries will be executed in the context of the same transaction.

Note: Python driver for SQLite doesn't start a transaction on a SELECT statement. It only begins a transaction on a statement which can modify the database: INSERT, UPDATE, DELETE. Other drivers start a transaction on any SQL statement, including SELECT.

A transaction ends when it is committed or rolled back using `commit()` or `rollback()` calls or implicitly by leaving the *db_session()* scope.

```
@db_session
def func():
    # a new transaction is started
    p = Product[123]
    p.price += 10
    # commit() will be done automatically
    # database session cache will be cleared automatically
    # database connection will be returned to the pool
```

Several transactions within the same db_session

If you need to have more than one transaction within the same database session you can call `commit()` or `rollback()` at any time during the session, and then the next query will start a new transaction. The Identity Map keeps data after the manual `commit()`, but if you call `rollback()` the cache will be cleared.

```
@db_session
def func1():
    p1 = Product[123]
    p1.price += 10
    commit()           # the first transaction is committed
    p2 = Product[456] # a new transaction is started
    p2.price -= 10
```

Nested db_session

If you enter the *db_session()* scope recursively, for example by calling a function which is decorated with the `@db_session` decorator from another function which is decorated with `@db_session()`, Pony will not create a new session, but will share the same session for both functions. The database session ends on leaving the scope of the outermost *db_session()* decorator or context manager.

db_session cache

Pony caches data at several stages for increasing performance. It caches:

- The results of a generator expression translation. If the same generator expression query is used several times within the program, it will be translated to SQL only once. This cache is global for entire program, not only for a single database session.
- Objects which were created or loaded from the database. Pony keeps these objects in the Identity Map. This cache is cleared on leaving the *db_session()* scope or on transaction rollback.
- Query results. Pony returns the query result from the cache if the same query is called with the same parameters once again. This cache is cleared once any of entity instances is changed. This cache is cleared on leaving the *db_session()* scope or on transaction rollback.

Using db_session with generator functions or coroutines

The `@db_session()` decorator can be used with generator functions or coroutines too. The generator function is the function that contains the `yield` keyword inside it. The coroutine is a function which is defined using the `async def` or decorated with `@asyncio.coroutine`.

If inside such a generator function or coroutine you'll try to use the `db_session` context manager, it will not work properly, because in Python context managers cannot intercept generator suspension. Instead, you need to wrap you generator function or coroutine with the `@db_session` decorator.

In other words, don't do this:

```
def my_generator(x):
    with db_session: # it won't work here!
        obj = MyEntity.get(id=x)
        yield obj
```

Do this instead:

```
@db_session
def my_generator( x ):
    obj = MyEntity.get(id=x)
    yield obj
```

With regular functions, the `@db_session()` decorator works as a scope. When your program leaves the *db_session()* scope, Pony finishes the transaction by performing commit (or rollback) and clears the db_session cache.

In case of a generator, the program can reenter the generator code for several times. In this case, when your program leaves the generator code, the db_session is not over, but suspended and Pony doesn't clear the cache. In the same time, we don't know if the program will come back to this generator code again. That is why you have to explicitly commit or rollback current transaction before the program leaves the generator on `yield`. On regular functions Pony calls `commit()` or `rollback()` automatically on leaving the *db_session()* scope.

In essence, here is the difference when using *db_session()* with generator functions:

1. You have to call `commit()` or `rollback()` before the `yield` expression explicitly.
2. Pony doesn't clear the transaction cache, so you can continue using loaded objects when coming back to the same generator.
3. With a generator function, the *db_session()* can be used only as a decorator, not a context manager. This is because in Python the context manager cannot understand that it was left on `yield`.
4. The *db_session()* parameters, such as `retry`, `serializable` cannot be used with generator functions. The only parameter that can be used in this case is `immediate`.

Parameters of db_session

As it was mentioned above *db_session()* can be used as a decorator or a context manager. It can receive parameters which are described in the *API Reference*.

Working with multiple databases

Pony can work with several databases simultaneously. In the example below we use PostgreSQL for storing user information and MySQL for storing information about addresses:

```
db1 = Database()

class User(db1.Entity):
    ...

db1.bind('postgres', ...)


db2 = Database()

class Address(db2.Entity):
    ...

db2.bind('mysql', ...)

@db_session
def do_something(user_id, address_id):
    u = User[user_id]
    a = Address[address_id]
    ...
```

On exiting from the `do_something()` function Pony will perform `commit()` or `rollback()` to both databases. If you need to commit to one database before exiting from the function you can use `db1.commit()` or `db2.commit()` methods.

Functions for working with transactions

There are three upper level functions that you can use for working with transactions:

- *commit()*
- *rollback()*
- *flush()*

Also there are three corresponding functions of the *Database* object:

- *Database.commit()*
- *Database.rollback()*
- *Database.flush()*

If you work with one database, there is no difference between using an upper level or the *Database* object methods.

Optimistic concurrency control

By default Pony uses the optimistic concurrency control concept for increasing performance. With this concept, Pony doesn't acquire locks on database rows. Instead it verifies that no other transaction has modified the data it has read or is trying to modify. If the check reveals conflicting modifications, the committing transaction gets the exception `OptimisticCheckError, 'Object XYZ was updated outside of current transaction'` and rolls back.

What should we do with this situation? First of all, this behavior is normal for databases which implement the MVCC pattern (e.g. Postgres, Oracle). For example, in Postgres, you will get the following error when a concurrent transaction changed the same data:

```
ERROR:  could not serialize access due to concurrent update
```

The current transaction rolls back, but it can be restarted. In order to restart the transaction automatically, you can use the `retry` parameter of the *db_session()* decorator (see more details about it later in this chapter).

How Pony does the optimistic check? For this purpose Pony tracks access to attributes of each object. If the user's code reads or modifies an object's attribute, Pony then will check if this attribute value remains the same in the database on commit. This approach guarantees that there will be no lost updates, the situation when during the current transaction another transaction changed the same object and then our transaction overrides the data without knowing there were changes.

During the optimistic check Pony verifies only those attributes which were read or written by the user. Also when Pony updates an object, it updates only those attributes which were changed by the user. This way it is possible to have two concurrent transactions which change different attributes of the same object and both of them succeed.

Generally the optimistic concurrency control increases the performance because transactions can complete without the expense of managing locks or without having transactions wait for other transactions' lock to clear. This approach shows very good results when conflicts are rare and our application reads data more often then writes.

However, if contention for writing data is frequent, the cost of repeatedly restarting transactions hurts performance. In this case the pessimistic locking can be more appropriate.

Pessimistic locking

Sometimes we need to lock an object in the database in order to prevent other transactions from modifying the same record. Within the database such a lock should be done using the SELECT FOR UPDATE query. In order to generate such a lock using Pony you should call the *Query.for_update()* method:

```
select(p for p in Product if p.price > 100).for_update()
```

The query above selects all instances of Product with the price greater than 100 and locks the corresponding rows in the database. The lock will be released upon commit or rollback of current transaction.

If you need to lock a single object, you can use the `get_for_update` method of an entity:

```
Product.get_for_update(id=123)
```

When you trying to lock an object using *for_update()* and it is already locked by another transaction, your request will need to wait until the row-level lock is released. To prevent the operation from waiting for other transactions to commit, use the `nowait=True` option:

```
select(p for p in Product if p.price > 100).for_update(nowait=True)
# or
Product.get_for_update(id=123, nowait=True)
```

In this case, if a selected row(s) cannot be locked immediately, the request reports an error, rather than waiting.

The main disadvantage of pessimistic locking is performance degradation because of the expense of database locks and limiting concurrency.

How Pony avoids lost updates

Lower isolation levels increase the ability of many users to access data at the same time, but it also can lead to database anomalies such as lost updates.

Let's consider an example. Say we have two accounts. We need to provide a function which can transfer money from one account to another. During the transfer we check if the account has enough funds.

Let's say we are using Django ORM for this task. Below if one of the possible ways of implementing such a function:

```
@transaction.atomic
def transfer_money(account_id1, account_id2, amount):
    account1 = Account.objects.get(pk=account_id1)
    account2 = Account.objects.get(pk=account_id2)
    if amount > account1.amount:    # validation
        raise ValueError("Not enough funds")
    account1.amount -= amount
    account1.save()
    account2.amount += amount
    account2.save()
```

By default in Django, each `save()` is performed in a separate transaction. If after the first `save()` there will be a failure, the amount will just disappear. Even if there will be no failure, if another transaction will try to get the account statement in between of two `save()` operations, the result will be wrong. In order to avoid such problems, both operations should be combined in one transaction. We can do that by decorating the function with the `@transaction.atomic` decorator.

But even in this case we can encounter a problem. If two bank branches will try to transfer the full amount to different accounts at the same time, both operations will be performed. Each function will pass the validation and finally one transaction will override the results of another one. This anomaly is called "lost update".

There are three ways to prevent such anomaly:

- Use the SERIALIZABLE isolation level
- Use SELECT FOR UPDATE instead SELECT
- Use optimistic checks

If you use the SERIALIZABLE isolation level, the database will not allow to commit the second transaction by throwing an exception during commit. The disadvantage of such approach is that this level requires more system resources.

If you use SELECT FOR UPDATE then the transaction which hits the database first will lock the row and another transaction will wait.

The optimistic check doesn't require more system resources and doesn't lock the database rows. It eliminates the lost update anomaly by ensuring that the data wasn't changed between the moment when we read it from the database and the commit operation.

The only way to avoid the lost update anomaly in Django is using the SELECT FOR UPDATE and you should use it explicitly. If you forget to do that or if you don't realize that the problem of lost update exists with your business logic, your data can be lost.

Pony allows using all three approaches, having the third one, optimistic checks, turned on by default. This way Pony avoids the lost update anomaly completely. Also using the optimistic checks allows the highest concurrency because it doesn't lock the database and doesn't require extra resources.

The similar function for transferring money would look this way in Pony:

The SERIALIZABLE approach:

```
@db_session(serializable=True)
def transfer_money(account_id1, account_id2, amount):
    account1 = Account[account_id1]
    account2 = Account[account_id2]
    if amount > account1.amount:
        raise ValueError("Not enough funds")
    account1.amount -= amount
    account2.amount += amount
```

The SELECT FOR UPDATE approach:

```
@db_session
def transfer_money(account_id1, account_id2, amount):
    account1 = Account.get_for_update(id=account_id1)
    account2 = Account.get_for_update(id=account_id2)
    if amount > account1.amount:
        raise ValueError("Not enough funds")
    account1.amount -= amount
    account2.amount += amount
```

The optimistic check approach:

```
@db_session
def transfer_money(account_id1, account_id2, amount):
    account1 = Account[account_id1]
    account2 = Account[account_id2]
    if amount > account1.amount:
```

```
        raise ValueError("Not enough funds")
    account1.amount -= amount
    account2.amount += amount
```

The last approach is used by default in Pony and you don't need to add anything else explicitly.

Transaction isolation levels and database peculiarities

See the *API Reference* for more details on this topic.

CHAPTER 8

Working with entity instances

Creating an entity instance

Creating an entity instance in Pony is similar to creating a regular object in Python:

```
customer1 = Customer(login="John", password="***",
                     name="John", email="john@google.com")
```

When creating an object in Pony, all the parameters should be specified as keyword arguments. If an attribute has a default value, you can omit it.

All created instances belong to the current *db_session()*. In some object-relational mappers, you are required to call an object's `save()` method in order to save it. This is inconvenient, as a programmer must track which objects were created or updated, and must not forget to call the `save()` method on each object.

Pony tracks which objects were created or updated and saves them in the database automatically when current *db_session()* is over. If you need to save newly created objects before leaving the *db_session()* scope, you can do so by using the *flush()* or *commit()* functions.

Loading objects from the database

Getting an object by primary key

The simplest case is when we want to retrieve an object by its primary key. To accomplish this in Pony, the user simply needs to put the primary key in square brackets, after the class name. For example, to extract a customer with the primary key value of 123, we can write:

```
customer1 = Customer[123]
```

The same syntax also works for objects with composite keys; we just need to list the elements of the composite primary key, separated by commas, in the same order that the attributes were defined in the entity class description:

```
order_item = OrderItem[order1, product1]
```

Pony raises the `ObjectNotFound` exception if object with such primary key doesn't exist.

Getting one object by unique combination of attributes

If you want to retrieve one object not by its primary key, but by another combination of attributes, you can use the *get()* method of an entity. In most cases, it is used for getting an object by the secondary unique key, but it can also be used to search by any other combination of attributes. As a parameter of the *get()* method, you need to specify the names of the attributes and their values. For example, if you want to receive a product under the name "Product 1", and you believe that database has only one product under this name, you can write:

```
product1 = Product.get(name='Product1')
```

If no object is found, *get()* returns `None`. If multiple objects are found, `MultipleObjectsFoundError` exception is raised.

You may want to use the *get()* method with primary key when we want to get `None` instead of `ObjectNotFound` exception if the object does not exists in database.

Method *get()* can also receive a lambda function as a single positioning argument. This method returns an instance of an entity, and not an object of the *Query* class.

Getting several objects

In order to retrieve several objects from a database, you should use the *select()* method of an entity. Its argument is a lambda function, which has a single parameter, symbolizing an instance of an object in the database. Inside this function, you can write conditions, by which you want to select objects. For example, if you want to find all products with the price higher than 100, you can write:

```
products = Product.select(lambda p: p.price > 100)
```

This lambda function will not be executed in Python. Instead, it will be translated into the following SQL query:

```
SELECT "p"."id", "p"."name", "p"."description",
       "p"."picture", "p"."price", "p"."quantity"
FROM "Product" "p"
WHERE "p"."price" > 100
```

The *select()* method returns an instance of the *Query* class. If you start iterating over this object, the SQL query will be sent to the database and you will get the sequence of entity instances. For example, this is how you can print out all product names and it's price:

```
for p in Product.select(lambda p: p.price > 100):
    print(p.name, p.price)
```

If you don't want to iterate over a query, but need just to get a list of objects, you can do so this way:

```
product_list = Product.select(lambda p: p.price > 100)[:]
```

Here we get a full slice `[:]` from the query. This is an equivalent of converting a query to a list:

```
product_list = list(Product.select(lambda p: p.price > 100))
```

Using parameters in queries

You can use variables in queries. Pony will pass those variables as parameters to the SQL query. One important advantage of declarative query syntax in Pony is that it offers full protection from SQL-injections, as all external parameters will be properly escaped.

Here is the example:

```
x = 100
products = Product.select(lambda p: p.price > x)
```

The SQL query which will be generated will look this way:

```
SELECT "p"."id", "p"."name", "p"."description",
       "p"."picture", "p"."price", "p"."quantity"
FROM "Product" "p"
WHERE "p"."price" > ?
```

This way the value of `x` will be passed as the SQL query parameter, which completely eliminates the risk of SQL-injection.

Sorting query results

If you need to sort objects in a certain order, you can use the *Query.order_by()* method.

```
Product.select(lambda p: p.price > 100).order_by(desc(Product.price))
```

In this example, we display names and prices of all products with price higher than 100 in a descending order.

The methods of the *Query* object modify the SQL query which will be sent to the database. Here is the SQL generated for the previous example:

```
SELECT "p"."id", "p"."name", "p"."description",
       "p"."picture", "p"."price", "p"."quantity"
FROM "Product" "p"
WHERE "p"."price" > 100
ORDER BY "p"."price" DESC
```

The *Query.order_by()* method can also receive a lambda function as a parameter:

```
Product.select(lambda p: p.price > 100).order_by(lambda p: desc(p.price))
```

Using the lambda function inside the .. code-block:: python method allows using advanced sorting expressions. For example, this is how you can sort our customers by the total price of their orders in the descending order:

```
Customer.select().order_by(lambda c: desc(sum(c.orders.total_price)))
```

In order to sort the result by several attributes, you need to separate them by a comma. For example, if you want to sort products by price in descending order, while displaying products with similar prices in alphabetical order, you can do it this way:

```
Product.select(lambda p: p.price > 100).order_by(desc(Product.price), Product.name)
```

The same query, but using lambda function will look this way:

```
Product.select(lambda p: p.price > 100).order_by(lambda p: (desc(p.price), p.name))
```

Note that according to Python syntax, if you return more than one element from lambda, you need to put them into parenthesis.

Limiting the number of selected objects

It is possible to limit the number of objects returned by a query by using the *Query.limit()* method, or by more compact Python slice notation. For example, this is how you can get the ten most expensive products:

```
Product.select().order_by(lambda p: desc(p.price))[:10]
```

The result of a slice is not a query object, but a final list of entity instances.

You can also use the *Query.page()* method as a convenient way of pagination the query results:

```
Product.select().order_by(lambda p: desc(p.price)).page(1)
```

Traversing relationships

In Pony you can traverse object relationships:

```
order = Order[123]
customer = order.customer
print customer.name
```

Pony tries to minimize the number of queries sent to the database. In the example above, if the requested `Customer` object was already loaded to the cache, Pony will return the object from the cache without sending a query to the database. But, if an object was not loaded yet, Pony still will not send a query immediately. Instead, it will create a "seed" object first. The seed is an object which has only the primary key initialized. Pony does not know how this object will be used, and there is always the possibility that only the primary key is needed.

In the example above, Pony get the object from database in the third line in, when accessing the `name` attribute. By using the "seed" concept, Pony achieves high efficiency and solves the "N+1" problem, which is a weakness of many other mappers.

Traversing is possible in the "to-many" direction as well. For example, if you have a `Customer` object and you loop through its `orders` attribute, you can do it this way:

```
c = Customer[123]
for order in c.orders:
    print order.state, order.price
```

Updating an object

When you assign new values to object attributes, you don't need to save each updated object manually. Changes will be saved in the database automatically on leaving the *db_session()* scope.

For example, in order to increase the number of products by 10 with a primary key of 123, you can use the following code:

```
Product[123].quantity += 10
```

For changing several attributes of the same object, you can do so separately:

```
order = Order[123]
order.state = "Shipped"
order.date_shipped = datetime.now()
```

or in a single line, using the *set()* method of an entity instance:

```
order = Order[123]
order.set(state="Shipped", date_shipped=datetime.now())
```

The *set()* method can be convenient when you need to update several object attributes at once from a dictionary:

```
order.set(**dict_with_new_values)
```

If you need to save the updates to the database before the current database session is finished, you can use the *flush()* or *commit()* functions.

Pony always saves the changes accumulated in the *db_session()* cache automatically before executing the following methods: `select()`, `get()`, `exists()`, `execute()` and `commit()`.

In future, Pony is going to support bulk update. It will allow updating multiple objects on the disk without loading them to the cache:

```
update(p.set(price=price * 1.1) for p in Product
                                if p.category.name == "T-Shirt")
```

Deleting an object

When you call the *delete()* method of an entity instance, Pony marks the object as deleted. The object will be removed from the database during the following commit.

For example, this is how we can delete an order with the primary key equal to 123:

```
Order[123].delete()
```

Bulk delete

Pony supports bulk delete for objects using the *delete()* function. This way you can delete multiple objects without loading them to the cache:

```
delete(p for p in Product if p.category.name == 'SD Card')
#or
Product.select(lambda p: p.category.name == 'SD Card').delete(bulk=True)
```

Note: Be careful with the bulk delete:

- *before_delete()* and *after_delete()* hooks will not be called on deleted objects.
- If an object was loaded into memory, it will not be removed from the *db_session()* cache on bulk delete.

Cascade delete

When Pony deletes an instance of an entity it also needs to delete its relationships with other objects. The relationships between two objects are defined by two relationship attributes. If another side of the relationship is declared as a `Set`, then we just need to remove the object from that collection. If another side is declared as `Optional`, then we need to set it to `None`. If another side is declared as `Required`, we cannot just assign `None` to that relationship attribute. In this case, Pony will try to do a cascade delete of the related object.

This default behavior can be changed using the *cascade_delete* option of an attribute. By default this option is set to `True` when another side of the relationship is declared as `Required` and `False` for all other relationship types.

If the relationship is defined as `Required` at the other end and `cascade_delete=False` then Pony raises the `ConstraintError` exception on deletion attempt.

Let's consider a couple of examples.

The example below raises the `ConstraintError` exception on an attempt to delete a group which has related students:

```
class Group(db.Entity):
    major = Required(str)
    items = Set("Student", cascade_delete=False)

class Student(db.Entity):
    name = Required(str)
    group = Required(Group)
```

In the following example, if a `Person` object has a related `Parssport` object, then if you'll try to delete the `Person` object, the `Passport` object will be deleted as well due to cascade delete:

```
class Person(db.Entity):
    name = Required(str)
    passport = Optional("Passport", cascade_delete=True)

class Passport(db.Entity):
    number = Required(str)
    person = Required("Person")
```

Saving objects in the database

Normally you don't need to bother of saving your entity instances in the database manually - Pony automatically commits all changes to the database on leaving the *db_session()* context. It is very convenient. In the same time, in some cases you might want to *flush()* or *commit()* data in the database before leaving the current database session.

If you need to get the primary key value of a newly created object, you can do *commit()* manually within the *db_session()* in order to get this value:

```
class Customer(db.Entity):
    id = PrimaryKey(int, auto=True)
    email = Required(str)

@db_session
def handler(email):
    c = Customer(email=email)
```

```
    # c.id is equal to None
    # because it is not assigned by the database yet
    commit()
    # the new object is persisted in the database
    # c.id has the value now
    print(c.id)
```

Order of saving objects

Usually Pony saves objects in the database in the same order as they are created or modified. In some cases Pony can reorder SQL INSERT statements if this is required for saving objects. Let's consider the following example:

```
from pony.orm import *

db = Database()

class TeamMember(db.Entity):
    name = Required(str)
    team = Optional('Team')

class Team(db.Entity):
    name = Required(str)
    team_members = Set(TeamMember)

db.bind('sqlite', ':memory:')
db.generate_mapping(create_tables=True)
sql_debug(True)

with db_session:
    john = TeamMember(name='John')
    mary = TeamMember(name='Mary')
    team = Team(name='Tenacity', team_members=[john, mary])
```

In the example above we create two team members and then a team object, assigning the team members to the team. The relationship between TeamMember and Team objects is represented by a column in the TeamMember table:

```
CREATE TABLE "Team" (
  "id" INTEGER PRIMARY KEY AUTOINCREMENT,
  "name" TEXT NOT NULL
)

CREATE TABLE "TeamMember" (
  "id" INTEGER PRIMARY KEY AUTOINCREMENT,
  "name" TEXT NOT NULL,
  "team" INTEGER REFERENCES "Team" ("id")
)
```

When Pony creates `john`, `mary` and `team` objects, it understands that it should reorder SQL INSERT statements and create an instance of the `Team` object in the database first, because it will allow using the team id for saving TeamMember rows:

```
INSERT INTO "Team" ("name") VALUES (?)
[u'Tenacity']

INSERT INTO "TeamMember" ("name", "team") VALUES (?, ?)
[u'John', 1]
```

```
INSERT INTO "TeamMember" ("name", "team") VALUES (?, ?)
[u'Mary', 1]
```

Cyclic chains during saving objects

Now let's say we want to have an ability to assign a captain to a team. For this purpose we need to add a couple of attributes to our entities: `Team.captain` and reverse attribute `TeamMember.captain_of`

```
class TeamMember(db.Entity):
    name = Required(str)
    team = Optional('Team')
    captain_of = Optional('Team')

class Team(db.Entity):
    name = Required(str)
    team_members = Set(TeamMember)
    captain = Optional(TeamMember, reverse='captain_of')
```

And here is the code for creating entity instances with a captain assigned to the team:

```
with db_session:
    john = TeamMember(name='John')
    mary = TeamMember(name='Mary')
    team = Team(name='Tenacity', team_members=[john, mary], captain=mary)
```

When Pony tries to execute the code above it raises the following exception:

```
pony.orm.core.CommitException: Cannot save cyclic chain: TeamMember -> Team ->␣
↪TeamMember
```

Why did it happen? Let's see. Pony sees that for saving the `john` and `mary` objects in the database it needs to know the id of the team, and tries to reorder the insert statements. But for saving the `team` object with the `captain` attribute assigned, it needs to know the id of `mary` object. In this case Pony cannot resolve this cyclic chain and raises an exception.

In order to save such a cyclic chain, you have to help Pony by adding the *`flush()`* command:

```
with db_session:
    john = TeamMember(name='John')
    mary = TeamMember(name='Mary')
    flush() # saves objects created by this moment in the database
    team = Team(name='Tenacity', team_members=[john, mary], captain=mary)
```

In this case, Pony will save the `john` and `mary` objects in the database first and then will issue SQL UPDATE statement for building the relationship with the `team` object:

```
INSERT INTO "TeamMember" ("name") VALUES (?)
[u'John']

INSERT INTO "TeamMember" ("name") VALUES (?)
[u'Mary']

INSERT INTO "Team" ("name", "captain") VALUES (?, ?)
[u'Tenacity', 2]
```

```
UPDATE "TeamMember"
SET "team" = ?
WHERE "id" = ?
[1, 2]

UPDATE "TeamMember"
SET "team" = ?
WHERE "id" = ?
[1, 1]
```

Entity methods

See the *Entity methods* part of the API Reference for details.

Entity hooks

See the *Entity hooks* part of the API Reference for details.

Serializing entity instances using pickle

Pony allows pickling entity instances, query results and collections. You might want to use it if you want to cache entity instances in an external cache (e.g. memcache). When Pony pickles entity instances, it saves all attributes except collections in order to avoid pickling a large set of data. If you need to pickle a collection attribute, you must pickle it separately. Example:

```
>>> from pony.orm.examples.estore import *
>>> products = select(p for p in Product if p.price > 100)[:]
>>> products
[Product[1], Product[2], Product[6]]
>>> import cPickle
>>> pickled_data = cPickle.dumps(products)
```

Now we can put the pickled data to a cache. Later, when we need our instances again, we can unpickle it:

```
>>> products = cPickle.loads(pickled_data)
>>> products
[Product[1], Product[2], Product[6]]
```

You can use pickling for storing objects in an external cache for improving application performance. When you unpickle objects, Pony adds them to the current *db_session()* as if they were just loaded from the database. Pony doesn't check if objects hold the same state in the database.

CHAPTER 9

Queries

Pony provides a very convenient way to query the database using the generator expression syntax. Pony allows programmers to work with objects which are stored in a database as if they were stored in memory, using native Python syntax. It makes development much easier.

For writing queries you can use Python generator expressions or lambdas.

Using Python generator expressions

A Python generator syntax is a very natural way of writing queries. Here is an example of a generator expression:

```
(expression for x in s if condition)
```

The meaning of this generator expression is the following:

```
for x in s:
    if condition:
        yield expression
```

Pony provides a number of functions, for example *select()*, that receive a generator expression, and then translate this generator into a SQL query. The process of the translation is described in this StackOverflow question.

Here is an example of a query:

```
select(c for c in Customer if sum(c.orders.total_price) > 1000)
```

This query returns an instance of a *Query* class, and you can then call the *Query* object methods for getting the result, for example:

```
select(c for c in Customer if sum(c.orders.total_price) > 1000).first()
```

Besides the list of entities, you can return either a list of attributes:

```
select(c.name for c in Customer if sum(c.orders.total_price) > 1000)
```

Or a list of tuples:

```
select((c, sum(c.orders.total_price)) for c in Customer
                                      if sum(c.orders.total_price) > 1000)
```

Using lambda functions

Instead of using a generator, you can write queries using the lambda function:

```
Customer.select(lambda c: sum(c.orders.price) > 1000)
```

From the point of the translation the query into SQL there is no difference, if you use a generator or a lambda. The only difference is that using the lambda you can only return entity instances - there is no way to return a list of specific entity attributes or a list of tuples.

Pony ORM functions used to query the database

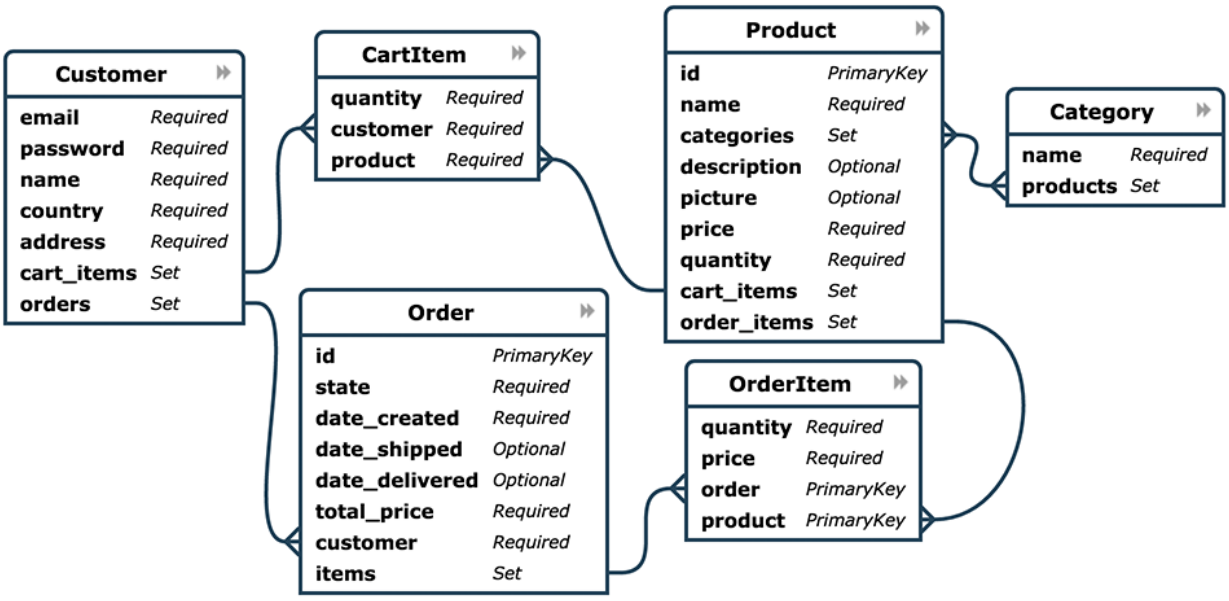
See the *Queries and functions* part of the API Reference for details.

Pony query examples

For demonstrating Pony queries let's use the example from the Pony ORM distribution. You can try these queries yourself in the interactive mode and see the generated SQL. For this purpose import the example module this way:

```
>>> from pony.orm.examples.estore import *
```

This module offers a simplified data model of a eCommerce online store. Here is the ER Diagram of the data model:



Here are the entity definitions:

```
from decimal import Decimal
from datetime import datetime

from pony.converting import str2datetime
from pony.orm import *

db = Database()

class Customer(db.Entity):
    email = Required(str, unique=True)
    password = Required(str)
    name = Required(str)
    country = Required(str)
    address = Required(str)
    cart_items = Set('CartItem')
    orders = Set('Order')

class Product(db.Entity):
    id = PrimaryKey(int, auto=True)
    name = Required(str)
    categories = Set('Category')
    description = Optional(str)
    picture = Optional(buffer)
    price = Required(Decimal)
    quantity = Required(int)
    cart_items = Set('CartItem')
    order_items = Set('OrderItem')

class CartItem(db.Entity):
    quantity = Required(int)
    customer = Required(Customer)
    product = Required(Product)

class OrderItem(db.Entity):
```

```
    quantity = Required(int)
    price = Required(Decimal)
    order = Required('Order')
    product = Required(Product)
    PrimaryKey(order, product)

class Order(db.Entity):
    id = PrimaryKey(int, auto=True)
    state = Required(str)
    date_created = Required(datetime)
    date_shipped = Optional(datetime)
    date_delivered = Optional(datetime)
    total_price = Required(Decimal)
    customer = Required(Customer)
    items = Set(OrderItem)

class Category(db.Entity):
    name = Required(str, unique=True)
    products = Set(Product)

sql_debug(True)
db.bind('sqlite', 'estore.sqlite', create_db=True)
db.generate_mapping(create_tables=True)
```

When you import this example, it will create the SQLite database in the file 'estore.sqlite' and fill it with some test data. Below you can see some query examples:

```
# All USA customers
Customer.select(lambda c: c.country == 'USA')

# The number of customers for each country
select((c.country, count(c)) for c in Customer)

# Max product price
max(p.price for p in Product)

# Max SSD price
max(p.price for p in Product
            for cat in p.categories if cat.name == 'Solid State Drives')

# Three most expensive products
Product.select().order_by(desc(Product.price))[:3]

# Out of stock products
Product.select(lambda p: p.quantity == 0)

# Most popular product
Product.select().order_by(lambda p: desc(sum(p.order_items.quantity))).first()

# Products that have never been ordered
Product.select(lambda p: not p.order_items)

# Customers who made several orders
Customer.select(lambda c: count(c.orders) > 1)

# Three most valuable customers
Customer.select().order_by(lambda c: desc(sum(c.orders.total_price)))[:3]
```

```
# Customers whose orders were shipped
Customer.select(lambda c: SHIPPED in c.orders.state)

# Customers with no orders
Customer.select(lambda c: not c.orders)

# The same query with the LEFT JOIN instead of NOT EXISTS
left_join(c for c in Customer for o in c.orders if o is None)

# Customers which ordered several different tablets
select(c for c in Customer
         for p in c.orders.items.product
         if 'Tablets' in p.categories.name and count(p) > 1)
```

You can find more queries in the pony.orm.examples.estore module.

Query object methods

See the *Query result* part of the API Reference for details.

Using date and time in queries

You can perform arithmetic operations with the `datetime` and `timedelta` in queries.

If the expression can be calculated in Python, Pony will pass the result of the calculation as a parameter into the query:

```
select(o for o in Order if o.date_created >= datetime.now() - timedelta(days=3))[:]
```

```
SELECT "o"."id", "o"."state", "o"."date_created", "o"."date_shipped",
       "o"."date_delivered", "o"."total_price", "o"."customer"
FROM "Order" "o"
WHERE "o"."date_created" >= ?
```

If the operation needs to be performed with the attribute, we cannot calculate it beforehand. That is why such expression will be translated into SQL:

```
select(o for o in Order if o.date_created + timedelta(days=3) >= datetime.now())[:]
```

```
SELECT "o"."id", "o"."state", "o"."date_created", "o"."date_shipped",
       "o"."date_delivered", "o"."total_price", "o"."customer"
FROM "Order" "o"
WHERE datetime("o"."date_created", '+3 days') >= ?
```

The SQL generated by Pony will vary depending on the database. Above is the example for SQLite. Here is the same query, translated into PostgreSQL:

```
SELECT "o"."id", "o"."state", "o"."date_created", "o"."date_shipped",
       "o"."date_delivered", "o"."total_price", "o"."customer"
FROM "order" "o"
WHERE ("o"."date_created" + INTERVAL '72:0:0' DAY TO SECOND) >= %(p1)s
```

If you need to use a SQL function, you can use the *raw_sql()* function in order to include this SQL fragment:

```
select(m for m in DBVoteMessage if m.date >= raw_sql("NOW() - '1 minute'::INTERVAL"))
```

With Pony you can use the `datetime` attributes, such as `month`, `hour`, etc. Depending on the database, it will be translated into different SQL, which extracts the value for this attribute. In this example we get the `month` attribute:

```
select(o for o in Order if o.date_created.month == 12)
```

Here is the result of the translation for SQLite:

```
SELECT "o"."id", "o"."state", "o"."date_created", "o"."date_shipped",
       "o"."date_delivered", "o"."total_price", "o"."customer"
FROM "Order" "o"
WHERE cast(substr("o"."date_created", 6, 2) as integer) = 12
```

And for PostgreSQL:

```
SELECT "o"."id", "o"."state", "o"."date_created", "o"."date_shipped",
       "o"."date_delivered", "o"."total_price", "o"."customer"
FROM "order" "o"
WHERE EXTRACT(MONTH FROM "o"."date_created") = 12
```

Automatic DISTINCT

Pony tries to avoid duplicates in a query result by automatically adding the `DISTINCT` SQL keyword where it is necessary, because useful queries with duplicates are very rare. When someone wants to retrieve objects with a specific criteria, they typically don't expect that the same object will be returned more than once. Also, avoiding duplicates makes the query result more predictable: you don't need to filter duplicates out of a query result.

Pony adds the `DISCTINCT` keyword only when there could be potential duplicates. Let's consider a couple of examples.

1. Retrieving objects with a criteria:

```
Person.select(lambda p: p.age > 20 and p.name == 'John')
```

In this example, the query doesn't return duplicates, because the result contains the primary key column of a Person. Since duplicates are not possible here, there is no need in the `DISTINCT` keyword, and Pony doesn't add it:

```
SELECT "p"."id", "p"."name", "p"."age"
FROM "Person" "p"
WHERE "p"."age" > 20
  AND "p"."name" = 'John'
```

2. Retrieving object attributes:

```
select(p.name for p in Person)
```

The result of this query returns not objects, but its attribute. This query result can contain duplicates, so Pony will add DISTINCT to this query:

```
SELECT DISTINCT "p"."name"
FROM "Person" "p"
```

The result of a such query typically used for a dropdown list, where duplicates are not expected. It is not easy to come up with a real use-case when you want to have duplicates here.

If you need to count persons with the same name, you'd better use an aggregate query:

```
select((p.name, count(p)) for p in Person)
```

But if it is absolutely necessary to get all person's names, including duplicates, you can do so by using the *Query.without_distinct()* method:

```
select(p.name for p in Person).without_distinct()
```

3. Retrieving objects using joins:

```
select(p for p in Person for c in p.cars if c.make in ("Toyota", "Honda"))
```

This query can contain duplicates, so Pony eliminates them using `DISTINCT`:

```
SELECT DISTINCT "p"."id", "p"."name", "p"."age"
FROM "Person" "p", "Car" "c"
WHERE "c"."make" IN ('Toyota', 'Honda')
  AND "p"."id" = "c"."owner"
```

Without using DISTINCT the duplicates are possible, because the query uses two tables (Person and Car), but only one table is used in the SELECT section. The query above returns only persons (and not their cars), and therefore it is typically not desirable to get the same person in the result more than once. We believe that without duplicates the result looks more intuitive.

But if for some reason you don't need to exclude duplicates, you always can add *without_distinct()* to the query:

```
select(p for p in Person for c in p.cars
         if c.make in ("Toyota", "Honda")).without_distinct()
```

The user probably would like to see the Person objects duplicates if the query result contains cars owned by each person. In this case the Pony query would be different:

```
select((p, c) for p in Person for c in p.cars if c.make in ("Toyota", "Honda"))
```

And in this case Pony will not add the `DISTINCT` keyword to SQL query.

To summarize:

1. The principle "all queries do not return duplicates by default" is easy to understand and doesn't lead to surprises.
2. Such behavior is what most users want in most cases.
3. Pony doesn't add DISTINCT when a query is not supposed to have duplicates.
4. The query method *without_distinct()* can be used for forcing Pony do not eliminate duplicates.

Functions which can be used inside a query

Here is the list of functions that can be used inside a generator query:

- *avg()*
- `abs()`
- *exists()*
- *len()*

- *max()*
- *min()*
- *count()*
- *concat()*
- *random()*
- *raw_sql()*
- *select()*
- *sum()*
- *getattr()*

Examples:

```
select(avg(c.orders.total_price) for c in Customer)
```

```
SELECT AVG("order-1"."total_price")
FROM "Customer" "c"
  LEFT JOIN "Order" "order-1"
    ON "c"."id" = "order-1"."customer"
```

```
select(o for o in Order if o.customer in
       select(c for c in Customer if c.name.startswith('A')))[:]
```

```
SELECT "o"."id", "o"."state", "o"."date_created", "o"."date_shipped",
       "o"."date_delivered", "o"."total_price", "o"."customer"
FROM "Order" "o"
WHERE "o"."customer" IN (
    SELECT "c"."id"
    FROM "Customer" "c"
    WHERE "c"."name" LIKE 'A%'
    )
```

Using getattr()

getattr() is a built-in Python function, that can be used for getting the attribute value.

Example:

```
attr_name = 'name'
param_value = 'John'
select(c for c in Customer if getattr(c, attr_name) == param_value)
```

Using raw SQL

Pony allows using raw SQL in your queries. There are two options on how you can use raw SQL:

1. Use the *raw_sql()* function in order to write only a part of a generator or lambda query using raw SQL.
2. Write a complete SQL query using the *Entity.select_by_sql()* or *Entity.get_by_sql()* methods.

Using the raw_sql() function

Let's explore examples of using the *raw_sql()* function. Here is the schema and initial data that we'll use for our examples:

```
from datetime import date
from pony.orm import *

db = Database('sqlite', ':memory:')

class Person(db.Entity):
    id = PrimaryKey(int)
    name = Required(str)
    age = Required(int)
    dob = Required(date)

db.generate_mapping(create_tables=True)

with db_session:
    Person(id=1, name='John', age=30, dob=date(1986, 1, 1))
    Person(id=2, name='Mike', age=32, dob=date(1984, 5, 20))
    Person(id=3, name='Mary', age=20, dob=date(1996, 2, 15))
```

The *raw_sql()* result can be treated as a logical expression:

```
select(p for p in Person if raw_sql('abs("p"."age") > 25'))
```

The *raw_sql()* result can be used for a comparison:

```
q = Person.select(lambda x: raw_sql('abs("x"."age")') > 25)
print(q.get_sql())

SELECT "x"."id", "x"."name", "x"."age", "x"."dob"
FROM "Person" "x"
WHERE abs("x"."age") > 25
```

Also, in the example above we use `raw_sql()` in a lambda query and print out the resulting SQL. As you can see the raw SQL part becomes a part of the whole query.

The *raw_sql()* can accept $parameters:

```
x = 25
select(p for p in Person if raw_sql('abs("p"."age") > $x'))
```

You can change the content of the *raw_sql()* function dynamically and still use parameters inside:

```
x = 1
s = 'p.id > $x'
select(p for p in Person if raw_sql(s))
```

Another way of using dynamic raw SQL content:

```
x = 1
cond = raw_sql('p.id > $x')
select(p for p in Person if cond)
```

You can use various types inside the raw SQL query:

```
x = date(1990, 1, 1)
select(p for p in Person if raw_sql('p.dob < $x'))
```

Parameters inside the raw SQL part can be combined:

```
x = 10
y = 15
select(p for p in Person if raw_sql('p.age > $(x + y)'))
```

You can even call Python functions inside:

```
select(p for p in Person if raw_sql('p.dob < $date.today()'))
```

The *raw_sql()* function can be used not only in the condition part, but also in the part which returns the result of the query:

```
names = select(raw_sql('UPPER(p.name)') for p in Person)[:]
print(names)

['JOHN', 'MIKE', 'MARY']
```

But when you return data using the *raw_sql()* function, you might need to specify the type of the result, because Pony has no idea on what the result type is:

```
dates = select(raw_sql('(p.dob)') for p in Person)[:]
print(dates)

['1985-01-01', '1983-05-20', '1995-02-15']
```

If you want to get the result as a list of dates, you need to specify the `result_type`:

```
dates = select(raw_sql('(p.dob)', result_type=date) for p in Person)[:]
print(dates)

[datetime.date(1986, 1, 1), datetime.date(1984, 5, 20), datetime.date(1996, 2, 15)]
```

The *raw_sql()* function can be used in a *Query.filter()* too:

```
x = 25
select(p for p in Person).filter(lambda p: p.age > raw_sql('$x'))
```

It can be used inside the *Query.filter()* without lambda. In this case you have to use the first letter of entity name in lower case as the alias:

```
x = 25
Person.select().filter(raw_sql('p.age > $x'))
```

You can use several *raw_sql()* expressions in a single query:

```
x = '123'
y = 'John'
Person.select(lambda p: raw_sql("UPPER(p.name) || $x")
                        == raw_sql("UPPER($y || '123')"))
```

The same parameter names can be used several times with different types and values:

```
x = 10
y = 31
q = select(p for p in Person if p.age > x and p.age < raw_sql('$y'))
x = date(1980, 1, 1)
y = 'j'
q = q.filter(lambda p: p.dob > x and p.name.startswith(raw_sql('UPPER($y)')))
persons = q[:]
```

You can use *raw_sql()* in *Query.order_by()* section:

```
x = 9
Person.select().order_by(lambda p: raw_sql('SUBSTR(p.dob, $x)'))
```

Or without lambda, if you use the same alias, that you used in previous filters. In this case we use the default alias - the first letter of the entity name:

```
x = 9
Person.select().order_by(raw_sql('SUBSTR(p.dob, $x)'))
```

Using the select_by_sql() and get_by_sql() methods

Although Pony can translate almost any condition written in Python to SQL, sometimes the need arises to use raw SQL, for example - in order to call a stored procedure or to use a dialect feature of a specific database system. In this case, Pony allows the user to write a query in a raw SQL, by placing it inside the function *Entity.select_by_sql()* or *Entity.get_by_sql()* as a string:

```
Product.select_by_sql("SELECT * FROM Products")
```

Unlike the method *Entity.select()*, the method *Entity.select_by_sql()* does not return the *Query* object, but a list of entity instances.

Parameters are passed using the following syntax: "$name_variable" or "$(expression in Python)". For example:

```
x = 1000
y = 500
Product.select_by_sql("SELECT * FROM Product WHERE price > $x OR price = $(y * 2)")
```

When Pony encounters a parameter within a raw SQL query, it gets the variable value from the current frame (from globals and locals) or from the dictionaries which can be passed as parameters:

```
Product.select_by_sql("SELECT * FROM Product WHERE price > $x OR price = $(y * 2)",
                      globals={'x': 100}, locals={'y': 200})
```

Variables and more complex expressions specified after the `$` sign, will be automatically calculated and transferred into the query as parameters, which makes SQL-injection impossible. Pony automatically replaces $x in the query string with "?", "%S" or with other paramstyle, used in your database.

If you need to use the `$` sign in the query (for example, in the name of a system table), you have to write two `$` signs in succession: `$$`.

CHAPTER 10

Working with entity relationships

In Pony, an entity can relate to other entities through relationship attributes. Each relationship always has two ends, and is defined by two entity attributes:

```
class Person(db.Entity):
    cars = Set('Car')

class Car(db.Entity):
    owner = Optional(Person)
```

In the example above we've defined one-to-many relationship between the `Person` and `Car` entities using the `Person.cars` and `Car.owner` attributes.

Let's add a couple more data attributes to our entities:

```
from pony.orm import *

db = Database()

class Person(db.Entity):
    name = Required(str)
    cars = Set('Car')

class Car(db.Entity):
    make = Required(str)
    model = Required(str)
    owner = Optional(Person)

db.bind('sqlite', ':memory:')
db.generate_mapping(create_tables=True)
```

Now let's create instances of `Person` and `Car` entities:

```
>>> p1 = Person(name='John')
>>> c1 = Car(make='Toyota', model='Camry')
>>> commit()
```

Normally, in your program, you don't need to call the function *commit()* manually, because it should be called automatically by the *db_session()*. But when you work in the interactive mode, you never leave a *db_session()*, that is why we need to commit manually if we want to store data in the database.

Establishing a relationship

Right after we've created the instances `p1` and `c1`, they don't have an established relationship. Let's check the values of the relationship attributes:

```
>>> print c1.owner
None

>>> print p1.cars
CarSet([])
```

The attribute `cars` has an empty set.

Now let's establish a relationship between these two instances:

```
>>> c1.owner = p1
```

If we print the values of relationship attributes now, then we'll see the following:

```
>>> print c1.owner
Person[1]

>>> print p1.cars
CarSet([Car[1]])
```

When we assigned an owner to the `Car` instance, the `Person.cars` relationship attribute reflected the change immediately.

We also could establish a relationship by assigning the relationship attribute during the creation of the `Car` instance:

```
>>> p1 = Person(name='John')
>>> c1 = Car(make='Toyota', model='Camry', owner=p1)
```

In our example the attribute `owner` is optional, so we can assign a value to it at any time, either during the creation of the `Car` instance, or later.

Operations with collections

The attribute `Person.cars` is represented as a collection and hence we can use regular operations that applicable to collections: `add()`, `remove()`, `in`, `len()`, `clear()`.

You can add or remove relationships using the *Set.add()* and *Set.remove()* methods:

```
>>> p1.cars.remove(Car[1])
>>> print p1.cars
CarSet([])

>>> p1.cars.add(Car[1])
>>> print p1.cars
CarSet([Car[1]])
```

You can check if a collection contains an element:

```
>>> Car[1] in p1.cars
True
```

Or make sure that there is no such element in the collection:

```
>>> Car[1] not in p1.cars
False
```

Check the collection length:

```
>>> len(p1.cars)
1
```

If you need to create an instance of a car and assign it with a particular person instance, there are several ways to do it. One of the options is to call the *create()* method of the collection attribute:

```
>>> p1.cars.create(model='Toyota', make='Prius')
>>> commit()
```

Now we can check that a new `Car` instance was added to the `Person.cars` collection attribute of our instance:

```
>>> print p1.cars
CarSet([Car[2], Car[1]])
>>> p1.cars.count()
2
```

You can iterate over a collection attribute:

```
>>> for car in p1.cars:
...     print car.model

Toyota
Camry
```

Attribute lifting

In Pony, the collection attributes provides the attribute lifting capability: the collection obtains its items' attributes:

```
>>> show(Car)
class Car(Entity):
    id = PrimaryKey(int, auto=True)
    make = Required(str)
    model = Required(str)
    owner = Optional(Person)
>>> p1 = Person[1]
>>> print p1.cars.model
Multiset({u'Camry': 1, u'Prius': 1})
```

Here we print out the entity declaration using the *show()* function and then print the value of the `model` attribute of the `cars` relationship attribute. The `cars` attribute has all the attributes of the `Car` entity: `id`, `make`, `model` and `owner`. In Pony we call this a Multiset and it is implemented using a dictionary. The dictionary's key represents the value of the attribute - 'Camry' and 'Prius' in our example. And the dictionary's value shows how many times it encounters in this collection.

```
>>> print p1.cars.make
Multiset({u'Toyota': 2})
```

`Person[1]` has two Toyotas.

We can iterate over the multiset:

```
>>> for m in p1.cars.make:
...     print m
...
Toyota
Toyota
```

Collection attribute parameters

Here is the list options that you can apply to collection attributes:

- *cascade_delete*
- *columns*
- *lazy*
- *nplus1_threshold*
- *reverse*
- *reverse_columns*
- *table*

Example:

```
class Photo(db.Entity):
    tags = Set('Tag', table='photo_to_tag', column='tag_id')

class Tag(db.Entity):
    photos = Set(Photo)
```

Collection attribute queries and other methods

Starting with the release 0.6.1, Pony introduces queries for the relationship attributes.

You can use the following methods of the relationship attribute for retrieving data:

- *Set.select()*
- *Set.random()*
- *Set.page()*
- *Set.order_by()*
- *Set.load()*
- *Set.filter()*

See the *Collection attribute methods* part of th API Reference for more details.

Below you can find several examples of using these methods. We'll use the University schema for showing these queries, here are python entity definitions and Entity-Relationship diagram.

The example below selects all students with the `gpa` greater than 3 within the group 101:

```
g = Group[101]
g.students.filter(lambda student: student.gpa > 3)[:]
```

This query can be used for displaying the second page of group 101 student's list ordered by the `name` attribute:

```
g.students.order_by(Student.name).page(2, pagesize=3)
```

The same query can be also written in the following form:

```
g.students.order_by(lambda s: s.name).limit(3, offset=3)
```

The following query returns two random students from the group 101:

```
g.students.random(2)
```

And one more example. This query returns the first page of courses which were taken by `Student[1]` in the second semester, ordered by the course name:

```
s = Student[1]
s.courses.select(lambda c: c.semester == 2).order_by(Course.name).page(1)
```

CHAPTER 11

Aggregation

You can use the following five aggregate functions for declarative queries: *sum()*, *count()*, *min()*, *max()*, and *avg()*. Let's see some examples of simple queries using these functions.

Total GPA of students from group 101:

```
sum(s.gpa for s in Student if s.group.number == 101)
```

Number of students with a GPA above three:

```
count(s for s in Student if s.gpa > 3)
```

First name of a student, who studies philosophy, sorted alphabetically:

```
min(s.name for s in Student if "Philosophy" in s.courses.name)
```

Birth date of the youngest student in group 101:

```
max(s.dob for s in Student if s.group.number == 101)
```

Average GPA in department 44:

```
avg(s.gpa for s in Student if s.group.dept.number == 44)
```

Note: Although Python already has the standard functions `sum()`, `count()`, `min()`, and `max()`, Pony adds its own functions under the same names. Also, Pony adds its own *avg()* function. These functions are implemented in the `pony.orm` module and they can be imported from there either "by the star", or by its name.

The functions implemented in Pony expand the behavior of standard functions in Python; thus, if in a program these functions are used in their standard way, the import will not affect their behavior. But it also allows specifying a declarative query inside the function.

If one forgets to import these functions from the `pony.orm` package, then an error will appear upon use of the Python standard functions `sum()`, `count()`, `min()`, and `max()` with a declarative query as a parameter:

```
TypeError: Use a declarative query in order to iterate over entity
```

Aggregate functions can also be used inside a query. For example, if you need to find not only the birth date of the youngest student in the group, but also the student himself, you can write the following query:

```
select(s for s in Student if s.group.number == 101
          and s.dob == max(s.dob for s in Student
                                 if s.group.number == 101))
```

Or, for example, to get all groups with an average GPA above 4.5:

```
select(g for g in Group if avg(s.gpa for s in g.students) > 4.5)
```

This query can be shorter if we use Pony *attribute lifting* feature:

```
select(g for g in Group if avg(g.students.gpa) > 4.5)
```

Query object aggregate functions

You can call the aggregate methods of the *Query* object:

```
select(sum(s.gpa) for s in Student)
```

Is equal to the following query:

```
select(s.gpa for s in Student).sum()
```

Here is the list of the aggregate functions:

- *Query.avg()*
- *Query.count()*
- *Query.min()*
- *Query.max()*
- *Query.sum()*

Several aggregate functions in one query

SQL allows you including several aggregate functions in the same query. For example, we might want to receive both the lowest and the highest GPA for each group. In SQL, such a query would look like this:

```
SELECT s.group_number, MIN(s.gpa), MAX(s.gpa)
FROM Student s
GROUP BY s.group_number
```

This query will return the lowest and the highest GPA for each group. With Pony you can use the same approach:

```
select((s.group, min(s.gpa), max(s.gpa)) for s in Student)
```

Function count

Aggregate queries often need to calculate the quantity of something. Here is how we get the number of students in Group 101:

```
count(s for s in Student if s.group.number == 101)
```

The number of students in each group related to the department 44:

```
select((g, count(g.students)) for g in Group if g.dept.number == 44)
```

or this way:

```
select((s.group, count(s)) for s in Student if s.group.dept.number == 44)
```

In the first example the aggregate function *count()* receives a collection, and Pony will translate it into a subquery. (Actually, this subquery will be optimized by Pony and will be replaced with `LEFT JOIN`).

In the second example, the function *count()* receives a single object instead of a collection. In this case Pony will add a `GROUP BY` section to the SQL query and the grouping will be done on the `s.group` attribute.

If you use the *count()* function without arguments, this will be translated to SQL `COUNT(*)`. If you specify an argument, it will be translated to `COUNT(DISTINCT column)`.

Conditional count

There is another way of using the *count()* function. Let's assume that we want to get three numbers for each group:

- The number of students that have a GPA less than 3
- The number of students with GPA between 3 to 4
- The number of students with GPA higher than 4

The query can be constructed this way:

```
select((g, count(s for s in g.students if s.gpa <= 3),
           count(s for s in g.students if s.gpa > 3 and s.gpa <= 4),
           count(s for s in g.students if s.gpa > 4)) for g in Group)
```

Although this query will work, it is pretty long and not very effecive - each `count` will be translated into a separate subquery. For such situations, Pony provides a "conditional COUNT" syntax:

```
select((s.group, count(s.gpa <= 3),
                 count(s.gpa > 3 and s.gpa <= 4),
                 count(s.gpa > 4)) for s in Student)
```

This way, we put our condition into the *count()* function. This query will not have subqueries, which makes it more effective.

Note: The queries above are not entirely equivalent: if a group doesn't have any students, then the first query will select that group having zeros as the result of *count()*, while the second query simply will not select the group at all. This happens because the second query selects the rows from the table Student, and if the group doesn't have any students, then the table Student will not have any rows for this group.

If you want to get rows with zeros, then an effective SQL query should use the *left_join()* function:

```
left_join((g, count(s.gpa <= 3),
       count(s.gpa > 3 and s.gpa <= 4),
       count(s.gpa > 4)) for g in Group for s in g.students)
```

More sophisticated aggregate queries

Using Pony you can do even more complex grouping. For example, you can group by an attribute part:

```
select((s.dob.year, avg(s.gpa)) for s in Student)
```

The birth year in this case is not a distinct attribute – it is a part of the `dob` attribute.

You can have expressions inside the aggregate functions:

```
select((item.order, sum(item.price * item.quantity))
        for item in OrderItem if item.order.id == 123)
```

Here is another way of making the same query:

```
select((order, sum(order.items.price * order.items.quantity))
        for order in Order if order.id == 123)
```

In the second case, we use the *attribute lifting* concept. The expression `order.items.price` creates an array of prices, while `order.items.quantity` generates an array of quantities. As the result, in this example, we'll have the sum of quantity multiplied by the price for each order item.

Queries with HAVING

The `SELECT` statement has two different sections which are used for conditions: `WHERE` and `HAVING`. The `WHERE` section is used more often and contains conditions which will be applied to each row. If a query contains aggregate functions, such as `MAX` or `SUM`, the `SELECT` statement may also contain `GROUP BY` and `HAVING` sections. The conditions of the `HAVING` section are applied after grouping the SQL query results. Typically the conditions of the `HAVING` section always contain aggregate functions, while conditions in the `WHERE` section may only contain aggregate functions inside a subquery.

When you write a query which contains aggregate functions, Pony needs to determine if the resulting SQL will contain the `GROUP BY` and `HAVING` sections and where it should put each condition from the Python query. If a condition contains an aggregate function, Pony places the condition into the `HAVING` section. Otherwise it places the condition into the `WHERE` section.

Consider the following query, which returns the tuples (`Group`, count_of_students):

```
select((s.group, count(s)) for s in Student
       if s.group.dept.number == 44 and avg(s.gpa) > 4)
```

In this query we have two conditions. The first condition is `s.group.dept.number == 44`. Since it doesn't include an aggregate function, Pony will place this condition into the `WHERE` section. The second condition `avg(s.gpa) > 4` contains the aggregate function `avg` and will be placed into the `HAVING` section.

Another question is what columns Pony should add to the `GROUP BY` section. According to the SQL standard, any non-aggregated column which placed into the `SELECT` statement should be added to the `GROUP BY` section too. Let's consider the following query:

```
SELECT A, B, C, SUM(D), MAX(E), COUNT(F)
FROM T1
WHERE ...
GROUP BY ...
HAVING ...
```

According to the SQL standard, we need to include the columns `A`, `B` and `C` into the `GROUP BY` section, because these columns appear in the `SELECT` list and don't wrapped with any aggregate function. Pony does exactly this. If your aggregated Pony query returns a tuple with several expressions, any non-aggregated expression will be placed into the `GROUP BY` section. Let's consider the same Pony query again:

```
select((s.group, count(s)) for s in Student
       if s.group.dept.number == 44 and avg(s.gpa) > 4)
```

This query returns the tuples (`Group`, count_of_students). The first element of the tuple, the `Group` instance, is not aggregated, so it will be placed into the `GROUP BY` section:

```
SELECT "s"."group", COUNT(DISTINCT "s"."id")
FROM "Student" "s", "Group" "group-1"
WHERE "group-1"."dept" = 44
  AND "s"."group" = "group-1"."number"
GROUP BY "s"."group"
HAVING AVG("s"."gpa") > 4
```

The `s.group` expression was placed into the `GROUP BY` section, and the condition `avg(s.gpa) > 4` was placed into the `HAVING` section of the query.

Sometimes the condition which should be placed into the `HAVING` section contains some non-aggregated columns. Such columns will be added to the `GROUP BY` section, because according to the SQL standard it is forbidden to use a non-aggregated column inside the `HAVING` section, if it was not added to the `GROUP BY` list.

Another example:

```
select((item.order, item.order.total_price,
     sum(item.price * item.quantity))
     for item in OrderItem
     if item.order.total_price < sum(item.price * item.quantity))
```

This query has the following condition: `item.order.total_price < sum(item.price * item.quantity)`, which contains an aggregate function and should be added to the `HAVING` section. But the part `item.order.total_price` is not aggregated. Hence, it will be added to the `GROUP BY` section in order to satisfy the SQL requirements.

Aggregate functions in order by section

The aggregate functions can be used inside the *`Query.order_by()`* function. Here is an example:

```
select((s.group, avg(s.gpa)) for s in Student) \
        .order_by(lambda s: desc(avg(s.gpa)))
```

Another way of ordering by an aggregated value is specifying the position number inside the *`Query.order_by()`* method:

```
select((s.group, avg(s.gpa)) for s in Student).order_by(-2)
```

CHAPTER 12

API Reference

Contents

Database

Database class

class `Database`

The `Database` object manages database connections using a connection pool. It is thread safe and can be shared between all threads in your application. The `Database` object allows working with the database directly using SQL, but most of the time you will work with entities and let Pony generate SQL statements for makeing the corresponding changes in the database. You can work with several databases at the same time, having a separate `Database` object for each database, but each entity always belongs to one database.

`bind`(*provider*, **args*, ***kwargs*)

Bind entities to a database.

Parameters

- **`provider`** (*`str`*) – the name of the database provider. The database provider is a module which resides in the `pony.orm.dbproviders` package. It knows how to work with a particular database. After the database provider name you should specify parameters which will be passed to the `connect()` method of the corresponding DBAPI driver. Pony comes with the following providers: “sqlite”, “postgres”, “mysql”, “oracle”.
- **`args`** – parameters required by the database driver.
- **`kwargs`** – parameters required by the database driver.

During the `bind()` call, Pony tries to establish a test connection to the database. If the specified parameters are not correct or the database is not available, an exception will be raised. After the connection to

the database was established, Pony retrieves the version of the database and returns the connection to the connection pool.

The method can be called only once for a database object. All consequent calls of this method on the same database will raise the `TypeError('Database object was already bound to ... provider')` exception.

```
db.bind('sqlite', ':memory:')
db.bind('sqlite', 'filename', create_db=True)
db.bind('postgres', user='', password='', host='', database='')
db.bind('mysql', host='', user='', passwd='', db='')
db.bind('oracle', 'user/password@dsn')
```

commit()
Save all changes made within the current *db_session()* using the *flush()* method and commits the transaction to the database.

You can call `commit()` more than once within the same *db_session()*. In this case the *db_session()* cache keeps the cached objects after commits. The cache will be cleaned up when the *db_session()* is over or if the transaction will be rolled back.

create_tables()
Check the existing mapping and create tables for entities if they don't exist. Also, Pony checks if foreign keys and indexes exist and create them if they are missing.

This method can be useful if you need to create tables after they were deleted using the *drop_all_tables()* method. If you don't delete tables, you probably don't need this method, because Pony checks and creates tables during *generate_mapping()* call.

disconnect()
Close the database connection for the current thread if it was opened.

drop_all_tables(*with_all_data=False*)
Drop all tables which are related to the current mapping.

Parameters **with_all_data** (*bool*) – `False` means Pony drops tables only if none of them contain any data. In case at least one of them is not empty, the method will raise the `TableIsNotEmpty` exception without dropping any table. In order to drop tables with data you should set `with_all_data=True`.

drop_table(*table_name*, *if_exists=False*, *with_all_data=False*)
Drop the `table_name` table.

If you need to delete a table which is mapped to an entity, you can use the class method *drop_table()* of an entity.

Parameters

- **table_name** (*str*) – the name of the table to be deleted, case sensitive.
- **if_exists** (*bool*) – when `True`, it will not raise the `TableDoesNotExist` exception if there is no such table in the database.
- **with_all_data** (*bool*) – if the table is not empty the method will raise the `TableIsNotEmpty` exception.

Entity
This attribute represents the base class which should be inherited by all entities which are mapped to the particular database.

Example:

```
db = Database()

class Person(db.Entity):
    name = Required(str)
    age = Required(int)
```

execute(*sql*, *globals=None*, *locals=None*)
Execute SQL statement.

Before executing the provided SQL, Pony flushes all changes made within the current *db_session()* using the *flush()* method.

Parameters

- **sql** (*str*) – the SQL statement text.
- **globals** (*dict*) –
- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Returns a DBAPI cursor.

Example:

```
cursor = db.execute("""create table Person (
              id integer primary key autoincrement,
              name text,
              age integer
      )""")

name, age = "Ben", 33
cursor = db.execute("insert into Person (name, age) values ($name, $age)")
```

See *Raw SQL* section for more info.

exists(*sql*, *globals=None*, *locals=None*)
Check if the database has at least one row which satisfies the query.

Before executing the provided SQL, Pony flushes all changes made within the current *db_session()* using the *flush()* method.

Parameters

- **sql** (*str*) – the SQL statement text.
- **globals** (*dict*) –
- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Return type bool

Example:

```
name = 'John'
if db.exists("select * from Person where name = $name"):
    print "Person exists in the database"
```

flush()
Save the changes accumulated in the *db_session()* cache to the database. You may never have a need to call this method manually, because it will be done on leaving the *db_session()* automatically.

Pony always saves the changes accumulated in the cache automatically before executing the following methods: *get()*, *exists()*, *execute()*, *commit()*, *select()*.

generate_mapping(*check_tables=True*, *create_tables=False*)

Map declared entities to the corresponding tables in the database. Creates tables, foreign key references and indexes if necessary.

Parameters

- **check_tables** (*bool*) – when `True`, Pony makes a simple check that the table names and attribute names in the database correspond to entities declaration. It doesn't catch situations when the table has extra columns or when the type of a particular column doesn't match. Set it to `False` if you want to generate mapping and create tables for your entities later, using the method *create_tables()*.
- **create_tables** (*bool*) – create tables, foreign key references and indexes if they don't exist. Pony generates the names of the database tables and columns automatically, but you can override this behavior if you want. See more details in the *Mapping customization* section.

get(*sql*, *globals=None*, *locals=None*)

Select one row or just one value from the database.

The `get()` method assumes that the query returns exactly one row. If the query returns nothing then Pony raises `RowNotFound` exception. If the query returns more than one row, the exception `MultipleRowsFound` will be raised.

Before executing the provided SQL, Pony flushes all changes made within the current *db_session()* using the *flush()* method.

Parameters

- **sql** (*str*) – the SQL statement text.
- **globals** (*dict*) –
- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Returns a tuple or a value. If your request returns a lot of columns then you can assign the resulting tuple of the `get()` method to a variable and work with it the same way as it is described in *select()* method.

Example:

```
id = 1
age = db.get("select age from Person where id = $id")

name, age = db.get("select name, age from Person where id = $id")
```

get_connection()

Return the active database connection. It can be useful if you want to work with the DBAPI interface directly. This is the same connection which is used by the ORM itself. The connection will be reset and returned to the connection pool on leaving the *db_session()* context or when the database transaction rolls back. This connection can be used only within the *db_session()* scope where the connection was obtained.

Returns a DBAPI connection.

global_stats

This attribute keeps the dictionary where the statistics for executed SQL queries is aggregated from all

threads. The key of this dictionary is the SQL statement and the value is an object of the *QueryStat* class.

insert(*table_name|entity*, *returning=None*, ***kwargs*)

Insert new rows into a table. This command bypasses the identity map cache and can be used in order to increase the performance when you need to create lots of objects and not going to read them in the same transaction. Also you can use the *execute()* method for this purpose. If you need to work with those objects in the same transaction it is better to create instances of entities and have Pony to save them in the database.

Parameters

- **table_name|entity** (*str*) – the name of the table where the data will be inserted. The name is case sensitive. Instead of the `table_name` you can use the `entity` class. In this case Pony will insert into the table associated with the `entity`.
- **returning** (*str*) – the name of the column that holds the automatically generated primary key. If you want the `insert()` method to return the value which is generated by the database, you should specify the name of the primary key column.
- **kwargs** (*dict*) – named parameters used within the query.

Example:

```
new_id = db.insert("Person", name="Ben", age=33, returning='id')
```

last_sql

Read-only attribute which keeps the text of the last SQL statement. It can be useful for debugging.

local_stats

This is a dictionary which keeps the SQL query statistics for the current thread. The key of this dictionary is the SQL statement and the value is an object of the *QueryStat* class.

merge_local_stats()

Merge the statistics from the current thread into the global statistics. You can call this method at the end of the HTTP request processing.

When you call this method, the value of *local_stats* will be merged to *global_stats*, and *local_stats* will be cleared.

In a web application, you can call this method on finishing processing an HTTP request. This way the *global_stats* attribute will contain the statistics for the whole application.

rollback()

Rolls back the current transaction and clears the *db_session()* cache.

select(*sql*, *globals=None*, *locals=None*)

Execute the SQL statement in the database and returns a list of tuples.

Parameters

- **sql** (*str*) – the SQL statement text.
- **globals** (*dict*) –
- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Returns a list of tuples.

Example:

```
result = select("select * from Person")
```

If a query returns more than one column and the names of table columns are valid Python identifiers, then you can access them as attributes:

```
for row in db.select("name, age from Person"):
    print row.name, row.age
```

Supported databases

SQLite

Using SQLite database is the easiest way to work with Pony because there is no need to install a database system separately - the SQLite database system is included in the Python distribution. It is a perfect choice for beginners who want to experiment with Pony in the interactive shell. In order to bind the *Database* object a SQLite database you can do the following:

```
db.bind('sqlite', filename, create_db=False)
```

db.**bind**(*'sqlite'*, *filename*, *create_db=False*)

Parameters

- **'sqlite'** (*str*) – tells Pony that we use the SQLite database.
- **filename** (*str*) – the name of the file where SQLite will store the data. The filename can be absolute or relative. If you specify a relative path, that path is appended to the directory path of the Python file where this database was created (and not to the current working directory). This is because sometimes a programmer doesn't have the control over the current working directory (e.g. in mod_wsgi application). This approach allows the programmer to create applications which consist of independent modules, where each module can work with a separate database. When working in the interactive shell, Pony requires that you to always specify the absolute path of the storage file.
- **create_db** (*bool*) – `True` means that Pony will try to create the database if such filename doesn't exists. If such filename exists, Pony will use this file.

Normally SQLite database is stored in a file on disk, but it also can be stored entirely in memory. This is a convenient way to create a SQLite database when playing with Pony in the interactive shell, but you should remember, that the entire in-memory database will be lost on program exit. Also you should not work with the same in-memory SQLite database simultaneously from several threads because in this case all threads share the same connection due to SQLite limitation.

In order to bind with an in-memory database you should specify `:memory:` instead of the filename:

```
db.bind('sqlite', ':memory:')
```

There is no need in the parameter `create_db` when creating an in-memory database.

Note: By default SQLite doesn't check foreign key constraints. Pony always enables the foreign key support by sending the command `PRAGMA foreign_keys = ON;` starting with the release 0.4.9.

PostgreSQL

Pony uses psycopg2 driver in order to work with PostgreSQL. In order to bind the `Database` object to PostgreSQL use the following line:

```
db.bind('postgres', user='', password='', host='', database='')
```

All the parameters that follow the Pony database provider name will be passed to the `psycopg2.connect()` method. Check the psycopg2.connect documentation in order to learn what other parameters you can pass to this method.

MySQL

```
db.bind('mysql', host='', user='', passwd='', db='')
```

Pony tries to use the MySQLdb driver for working with MySQL. If this module cannot be imported, Pony tries to use pymysql. See the MySQLdb and pymysql documentation for more information regarding these drivers.

Oracle

```
db.bind('oracle', 'user/password@dsn')
```

Pony uses the **cx_Oracle** driver for connecting to Oracle databases. More information about the parameters which you can use for creating a connection to Oracle database can be found here.

Transactions & db_session

@db_session(*allowed_exceptions=[], immediate=False, retry=0, retry_exceptions=[TransactionError], serializable=False, strict=False*)

Used for establishing a database session.

Parameters

- **allowed_exceptions** (*list*) – a list of exceptions which when occurred do not cause the transaction rollback. Can be useful with some web frameworks which trigger HTTP redirect with the help of an exception.
- **immediate** (*bool*) – tells Pony when start a transaction with the database. Some databases (e.g. SQLite, Postgres) start a transaction only when a modifying query is sent to the database(UPDATE, INSERT, DELETE) and don't start it for SELECTs. If you need to start a transaction on SELECT, then you should set `immediate=True`. Usually there is no need to change this parameter.
- **retry** (*int*) – specifies the number of attempts for committing the current transaction. This parameter can be used with the `@db_session` decorator only. The decorated function should not call `commit()` or `rollback()` functions explicitly. When this parameter is specified, Pony catches the `TransactionError` exception (and all its descendants) and restarts the current transaction. By default Pony catches the `TransactionError` exception only, but this list can be modified using the `retry_exceptions` parameter.
- **retry_exceptions** (*list|callable*) – a list of exceptions which will cause the transaction restart. By default this parameter is equal to `[TransactionError]`. Another option is using a callable which returns a boolean value. This callable receives the

only parameter - an exception object. If this callable returns `True` then the transaction will be restarted.

- **`serializable`** (*`bool`*) – allows setting the SERIALIZABLE isolation level for a transaction.
- **`strict`** (*`bool`*) – *Experimental* when `True` the cache will be cleared on exiting the `db_session`. If you'll try to access an object after the session is over, you'll get the `pony.orm.core.DatabaseSessionIsOver` exception. Normally Pony strongly advises that you work with entity objects only within the `db_session`. But some Pony users want to access extracted objects in read-only mode even after the `db_session` is over. In order to provide this feature, by default, Pony doesn't purge cache on exiting from the `db_session`. This might be handy, but in the same time, this can require more memory for keeping all objects extracted from the database in cache.

Can be used as a decorator or a context manager. When the session ends it performs the following actions:

•Commits transaction if data was changed and no exceptions occurred otherwise it rolls back transaction.

•Returns the database connection to the connection pool.

•Clears the Identity Map cache.

If you forget to specify the `db_session` where necessary, Pony will raise the `TransactionError: db_session is required when working with the database` exception.

When you work with Python's interactive shell you don't need to worry about the database session, because it is maintained by Pony automatically.

If you'll try to access instance's attributes which were not loaded from the database outside of the `db_session` scope, you'll get the `DatabaseSessionIsOver` exception. This happens because by this moment the connection to the database is already returned to the connection pool, transaction is closed and we cannot send any queries to the database.

When Pony reads objects from the database it puts those objects to the Identity Map. Later, when you update an object's attributes, create or delete an object, the changes will be accumulated in the Identity Map first. The changes will be saved in the database on transaction commit or before calling the following functions: *`get()`*, *`exists()`*, *`commit()`*, *`select()`*.

Example of usage as a decorator:

```
@db_session
def check_user(username):
    return User.exists(username=username)
```

As a context manager:

```
def process_request():
    ...
    with db_session:
        u = User.get(username=username)
        ...
```

Transaction isolation levels and database peculiarities

Isolation is a property that defines when the changes made by one transaction become visible to other concurrent transactions Isolation levels. The ANSI SQL standard defines four isolation levels:

- READ UNCOMMITTED - the most unsafe level

- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE - the most safe level

When using the SERIALIZABLE level, each transaction sees the database as a snapshot made at the beginning of a transaction. This level provides the highest isolation, but it requires more resources than other levels.

This is the reason why most databases use a lower isolation level by default which allow greater concurrency. By default Oracle and PostgreSQL use READ COMMITTED, MySQL - REPEATABLE READ. SQLite supports the SERIALIZABLE level only, but Pony emulates the READ COMMITTED level for allowing greater concurrency.

If you want Pony to work with transactions using the SERIALIZABLE isolation level, you can do that by specifying the `serializable=True` parameter to the *db_session()* decorator or *db_session()* context manager:

```
@db_session(serializable=True)
def your_function():
    ...
```

READ COMMITTED vs. SERIALIZABLE mode

In SERIALIZABLE mode, you always have a chance to get a "Can't serialize access due to concurrent update" error, and would have to retry the transaction until it succeeded. You always need to code a retry loop in your application when you are using SERIALIZABLE mode for a writing transaction.

In READ COMMITTED mode, if you want to avoid changing the same data by a concurrent transaction, you should use SELECT FOR UPDATE. But this way there is a chance to have a database deadlock - the situation where one transaction is waiting for a resource which is locked by another transaction. If your transaction got a deadlock, your application needs to restart the transaction. So you end up needing a retry loop either way. Pony can restart a transaction automatically if you specify the `retry` parameter to the *db_session()* decorator (but not the *db_session()* context manager):

```
@db_session(retry=3)
def your_function():
    ...
```

SQLite

When using SQLite, Pony's behavior is similar as with PostgreSQL: when a transaction is started, selects will be executed in the autocommit mode. The isolation level of this mode is equivalent of READ COMMITTED. This way the concurrent transactions can be executed simultaneously with no risk of having a deadlock (the `sqlite3.OperationalError: database is locked` is not arising with Pony ORM). When your code issues non-select statement, Pony begins a transaction and all following SQL statements will be executed within this transaction. The transaction will have the SERIALIZABLE isolation level.

PostgreSQL

PostgreSQL uses the READ COMMITTED isolation level by default. PostgreSQL also supports the autocommit mode. In this mode each SQL statement is executed in a separate transaction. When your application just selects data from the database, the autocommit mode can be more effective because there is no need to send commands for beginning and ending a transaction, the database does it automatically for you. From the isolation point of view, the autocommit mode is nothing different from the READ COMMITTED isolation level. In both cases your application sees the data which have been committed by this moment.

Pony automatically switches from the autocommit mode and begins an explicit transaction when your application needs to modify data by several INSERT, UPDATE or DELETE SQL statements in order to provide atomicity of data update.

MySQL

MySQL uses the REPEATABLE READ isolation level by default. Pony doesn't use the autocommit mode with MySQL because there is no benefit of using it here. The transaction begins with the first SQL statement sent to the database even if this is a SELECT statement.

Oracle

Oracle uses the READ COMMITTED isolation level by default. Oracle doesn't have the autocommit mode. The transaction begins with the first SQL statement sent to the database even if this is a SELECT statement.

Entity definition

An entity is a Python class which stores an object's state in the database. Each instance of an entity corresponds to a row in the database table. Often entities represent objects from the real world (e.g. Customer, Product).

Entity attributes

Entity attributes are specified as class attributes inside the entity class using the syntax:

```
class EntityName(inherits_from)
    attr_name = attr_kind(attr_type, attr_options)
```

For example:

```
class Person(db.Entity):
    id = PrimaryKey(int, auto=True)
    name = Required(str)
    age = Optional(int)
```

Attribute kinds

Each entity attribute can be one of the following kinds:

- `Required` - must have a value at all times
- `Optional` - the value is optional
- `PrimaryKey` - defines a primary key attribute
- `Set` - represents a collection, used for 'to-many' relationships
- `Discriminator` - used for entity inheritance

Optional string attributes

For most data types `None` is used when no value is assigned to the attribute. But when a string attribute is not assigned a value, Pony uses an empty string instead of `None`. This is more practical than storing empty string as `NULL` in the database. Most frameworks behave this way. Also, empty strings can be indexed for faster search, unlike NULLs. If you will try to assign `None` to such an optional string attribute, you'll get the `ConstraintError` exception.

You can change this behavior using the `nullable=True` option. In this case it will be possible to store both empty strings and `NULL` values in the same column, but this is rarely needed.

Oracle database treats empty strings as `NULL` values. Because of this all `Optional` attributes in Oracle have `nullable` set to `True` automatically.

If an optional string attribute is used as a unique key or as a part of a unique composite key, it will always have `nullable` set to `True` automatically.

Composite primary and secondary keys

Pony fully supports composite keys. In order to declare a composite primary key you need to specify all the parts of the key as `Required` and then combine them into a composite primary key:

```
class Example(db.Entity):
    a = Required(int)
    b = Required(str)
    PrimaryKey(a, b)
```

In order to declare a secondary composite key you need to declare attributes as usual and then combine them using the `composite_key` directive:

```
class Example(db.Entity):
    a = Required(str)
    b = Optional(int)
    composite_key(a, b)
```

In the database `composite_key(a, b)` will be represented as the `UNIQUE ("a", "b")` constraint.

Composite indexes

Using the `composite_index()` directive you can create a composite index for speeding up data retrieval. It can combine two or more attributes:

```
class Example(db.Entity):
    a = Required(str)
    b = Optional(int)
    composite_index(a, b)
```

The composite index can include a discriminator attribute used for inheritance.

Using the `composite_index()` you can create a non-unique index. In order to define an unique index, use the `composite_key()` function described above.

Attribute data types

Pony supports the following attribute types:

- str
- unicode
- int
- float
- Decimal
- datetime
- date
- time
- timedelta
- bool
- buffer - used for binary data in Python 2 and 3
- bytes - used for binary data in Python 3
- LongStr - used for large strings
- LongUnicode - used for large strings
- UUID
- Json - used for mapping to native database JSON type

Also you can specify another entity as the attribute type for defining a relationship between two entities.

Strings in Python 2 and 3

As you know, Python 3 has some differences from Python 2 when it comes to strings. Python 2 provides two string types – `str` (byte string) and `unicode` (unicode string), whereas in Python 3 the `str` type represents unicode strings and the `unicode` was just removed.

Before the release 0.6, Pony stored `str` and `unicode` attributes as unicode in the database, but for `str` attributes it had to convert unicode to byte string on reading from the database. Starting with the Pony Release 0.6 the attributes of `str` type in Python 2 behave as if they were declared as `unicode` attributes. There is no difference now if you specify `str` or `unicode` as the attribute type – you will have unicode string in Python and in the database.

Starting with the Pony Release 0.6, where the support for Python 3 was added, instead of `unicode` and `LongUnicode` we recommend to use `str` and `LongStr` types respectively. `LongStr` and `LongUnicode` are stored as CLOB in the database.

The same thing is with the `LongUnicode` and `LongStr`. `LongStr` now is an alias to `LongUnicode`. This type uses unicode in Python and in the database.

```
attr1 = Required(str)
# is the same as
attr2 = Required(unicode)

attr3 = Required(LongStr)
# is the same as
attr4 = Required(LongUnicode)
```

Buffer and bytes types in Python 2 and 3

If you need to represent byte sequence in Python 2, you can use the `buffer` type. In Python 3 you should use the `bytes` type for this purpose. `buffer` and `bytes` types are stored as binary (BLOB) types in the database.

In Python 3 the `buffer` type has gone, and Pony uses the `bytes` type which was added in Python 3 to represent binary data. But for the sake of backward compatibility we still keep `buffer` as an alias to the `bytes` type in Python 3. If you're importing `*` from `pony.orm` you will get this alias too.

If you want to write code which can run both on Python 2 and Python 3, you should use the `buffer` type for binary attributes. If your code is for Python 3 only, you can use `bytes` instead:

```
attr1 = Required(buffer)  # Python 2 and 3

attr2 = Required(bytes)  # Python 3 only
```

It would be cool if we could use the `bytes` type as an alias to `buffer` in Python 2, but unfortunately it is impossible, because Python 2.6 adds bytes as a synonym for the str type.

Attribute options

Attribute options can be specified as positional and as keyword arguments during an attribute definition.

Max string length

String types can accept a positional argument which specifies the max length of this column in the database:

```
class Person(db.Entity):
    name = Required(str, 40)    #  VARCHAR(40)
```

Decimal scale and precision

For the `Decimal` type you can specify precision and scale:

```
class Product(db.Entity):
    price = Required(Decimal, 10, 2)    #  DECIMAL(10, 2)
```

Datetime and time precision

The `datetime` and `time` types accept a positional argument which specifies the column's precision. By default it is equal to 6 for most databases.

For MySQL database the default value is 0. Before the MySQL version 5.6.4, the `DATETIME` and `TIME` columns were unable to store fractional seconds at all. Starting with the version 5.6.4, you can store fractional seconds if you set the precision equal to 6 during the attribute definition:

```
class Action(db.Entity):
    dt = Required(datetime, 6)
```

Keyword argument options

Additional attribute options can be set as keyword arguments. For example:

```
class Customer(db.Entity):
    email = Required(str, unique=True)
```

Below you can find the list of available options:

auto
(*bool*) Can be used for a PrimaryKey attribute only. If `auto=True` then the value for this attribute will be assigned automatically using the database's incremental counter or sequence.

autostrip
(*bool*) Automatically removes leading and trailing whitespace characters in a string attribute. Similar to Python `string.strip()` function. By default is `True`.

cascade_delete
(*bool*) Controls the cascade deletion of related objects. `True` means that Pony always does cascade delete even if the other side is defined as `Optional`. `False` means that Pony never does cascade delete for this relationship. If the relationship is defined as `Required` at the other end and `cascade_delete=False` then Pony raises the `ConstraintError` exception on deletion attempt. *See also*.

column
(*str*) Specifies the name of the column in the database table which is used for mapping. By default Pony uses the attribute name as the column name in the database.

columns
(*list*) Specifies the column names in the database table which are used for mapping a composite attribute.

default
(*numeric|str|function*) Allows specifying a default value for the attribute. Pony processes default values in Python, it doesn't add SQL DEFAULT clause to the column definition. This is because the default expression can be not only a constant, but any arbitrary Python function. For example:

```
import uuid
from pony.orm import *

db = Database()

class MyEntity(db.Entity):
    code = Required(uuid.UUID, default=uuid.uuid4)
```

If you need to set a default value in the database, you should use the `sql_default` option.

index
(*bool|str*) Allows to control index creation for this column. `index=True` - the index will be created with the default name. `index='index_name'` - create index with the specified name. `index=False` – skip index creation. If no 'index' option is specified then Pony still creates index for foreign keys using the default name.

lazy
(*bool*) When `True`, then Pony defers loading the attribute value when loading the object. The value will not be loaded until you try to access this attribute directly. By default `lazy` is set to `True` for `LongStr` and `LongUnicode` and to `False` for all other types.

max
(*numeric*) Allows specifying the maximum allowed value for numeric attributes (int, float, Decimal). If you will try to assign the value that is greater than the specified max value, you'll get the `ValueError` exception.

min
(*numeric*) Allows specifying the minimum allowed value for numeric attributes (int, float, Decimal). If you will try to assign the value that is less than the specified min value, you'll get the `ValueError` exception.

nplus1_threshold
(*int*) This parameter is used for fine tuning the threshold used for the N+1 problem solution.

nullable
(*bool*) `True` allows the column to be `NULL` in the database. Most likely you don't need to specify this option because Pony sets it to the most appropriate value by default.

py_check
(*function*) Allows to specify a function which will be used for checking the value before it is assigned to the attribute. The function should return `True` or `False`. Also it can raise the `ValueError` exception if the check failed.

```
class Student(db.Entity):
    name = Required(str)
    gpa = Required(float, py_check=lambda val: val >= 0 and val <= 5)
```

reverse
(*str*) Specifies the attribute name at the other end which should be used for the relationship. It might be needed if there are more than one relationship between two entities.

reverse_column
(*str*) Used for a symmetric relationship in order to specify the name of the database column for the intermediate table.

reverse_columns
(*str*) Used for a symmetric relationship if the entity has a composite primary key. Allows you to specify the name of the database columns for the intermediate table.

size
(*int*) For the `int` type you can specify the size of integer type that should be used in the database using the `size` keyword. This parameter receives the number of bits that should be used for representing an integer in the database. Allowed values are 8, 16, 24, 32 and 64:

```
attr1 = Required(int, size=8)   # 8 bit - TINYINT in MySQL
attr2 = Required(int, size=16)  # 16 bit - SMALLINT in MySQL
attr3 = Required(int, size=24)  # 24 bit - MEDIUMINT in MySQL
attr4 = Required(int, size=32)  # 32 bit - INTEGER in MySQL
attr5 = Required(int, size=64)  # 64 bit - BIGINT in MySQL
```

You can use the `unsigned` parameter to specify that the attribute is unsigned:

```
attr1 = Required(int, size=8, unsigned=True) # TINYINT UNSIGNED in MySQL
```

The default value of the `unsigned` parameter is `False`. If `unsigned` is set to `True`, but `size` is not provided, `size` assumed to be 32 bits.

If current database does not support specified attribute size, the next bigger size is used. For example, PostgreSQL does not have `MEDIUMINT` numeric type, so `INTEGER` type will be used for an attribute with size 24.

Only MySQL actually supports unsigned types. For other databases the column will use signed numeric type which can hold all valid values for the specified unsigned type. For example, in PostgreSQL an unsigned attribute with size 16 will use `INTEGER` type. An unsigned attribute with size 64 can be represented only in MySQL and Oracle.

When the size is specified, Pony automatically assigns `min` and `max` values for this attribute. For example, a signed attribute with size 8 will receive `min` value -128 and `max` value 127, while unsigned attribute with the same size will receive `min` value 0 and `max` value 255. You can override `min` and `max` with your own values if necessary, but these values should not exceed the range implied by the size.

Starting with the Pony release 0.6 the `long` type is deprecated and if you want to store 64 bit integers in the database, you need to use `int` instead with `size=64`. If you don't specify the `size` parameter, Pony will use the default integer type for the specific database.

sequence_name
(*str*) Allows to specify the sequence name used for `PrimaryKey` attributes. *Oracle database only.*

sql_default
(*str*) This option allows specifying the default SQL text which will be included to the CREATE TABLE SQL command. For example:

```
class MyEntity(db.Entity):
    created_at = Required(datetime, sql_default='CURRENT_TIMESTAMP')
    closed = Required(bool, default=True, sql_default='1')
```

Specifying `sql_default=True` can be convenient when you have a `Required` attribute and the value for it is going to be calculated in the database during the INSERT command (e.g. by a trigger). `None` by default.

sql_type
(*str*) Sets a specific SQL type for the column.

unique
(*bool*) If `True`, then the database will check that the value of this attribute is unique.

unsigned
(*bool*) Allows creating unsigned types in the database. Also checks that the assigned value is positive.

table
(*str*) Used for many-to-many relationship only in order to specify the name of the intermediate table.

volatile
(*bool*) Usually you specify the value of the attribute in Python and Pony stores this value in the database. But sometimes you might want to have some logic in the database which changes the value for a column. For example, you can have a trigger in the database which updates the timestamp of the last object's modification. In this case you want to have Pony to forget the value of the attribute on object's update sent to the database and read it from the database at the next access attempt. Set `volatile=True` in order to let Pony know that this attribute can be changed in the database.

The `volatile=True` option can be combined with the `sql_default` option if the value for this attribute is going to be both created and updated by the database.

You can get the exception `UnrepeatableReadError: Value ... was updated outside of current transaction` if another transaction changes the value of the attribute which is used in the current transaction. Pony notifies about it because this situation can break the business logic of the application. If you don't want Pony to protect you from such concurrent modifications you can set `volatile=True` for the attribute.

Collection attribute methods

To-many attributes have methods that provide a convenient way of querying data. You can treat a to-many relationship attribute as a regular Python collection and use standard operations like `in`, `not in`, `len`. Also Pony provides the following methods:

class Set

__len__()
Return the number of objects in the collection. If the collection is not loaded into cache, this methods loads all the collection instances into the cache first, and then returns the number of objects. Use this method if you are going to iterate over the objects and you need them loaded into the cache. If you don't need the collection to be loaded into the memory, you can use the *count()* method.

```
>>> p1 = Person[1]
>>> Car[1] in p1.cars
True
>>> len(p1.cars)
2
```

add(*item|iter*)
Add instances to a collection and establish a two-way relationship between entity instances:

```
photo = Photo[123]
photo.tags.add(Tag['Outdoors'])
```

Now the instance of the `Photo` entity with the primary key 123 has a relationship with the `Tag['Outdoors']` instance. The attribute `photos` of the `Tag['Outdoors']` instance contains the reference to the `Photo[123]` as well.

You can also establish several relationships at once passing the list of tags to the `add()` method:

```
photo.tags.add([Tag['Party'], Tag['New Year']])
```

clear()
Remove all items from the collection which means breaking relationships between entity instances.

copy()
Return a Python `set` object which contains the same items as the given collection.

count()
Return the number of objects in the collection. This method doesn't load the collection instances into the cache, but generates an SQL query which returns the number of objects from the database. If you are going to work with the collection objects (iterate over the collection or change the object attributes), you might want to use the *__len__()* method.

create(***kwargs*)
Create an return an instance of the related entity and establishes a relationship with it:

```
new_tag = Photo[123].tags.create(name='New tag')
```

is an equivalent of the following:

```
new_tag = Tag(name='New tag')
Photo[123].tags.add(new_tag)
```

drop_table(*with_all_data=False*)
Drop the intermediate table which is created for establishing many-to-many relationship. If the table is not empty and `with_all_data=False`, the method raises the `TableIsNotEmpty` exception and doesn't delete anything. Setting the `with_all_data=True` allows you to delete the table even if it is not empty.

```
class Product(db.Entity):
    tags = Set('Tag')

class Tag(db.Entity):
    products = Set(Product)

Product.tags.drop_table(with_all_data=True) # removes the intermediate table
```

is_empty()
Check if the collection is empty. Returns `False` if there is at lease one relationship and `True` if this attribute has no relationships.

filter()
Select objects from a collection. The method names *select()* and *filter()* are synonyms. Example:

```
g = Group[101]
g.students.filter(lambda student: student.gpa > 3)
```

load()
Load all related objects from the database.

order_by(*attr|lambda*)
Return an ordered collection.

```
g.students.order_by(Student.name).page(2, pagesize=3)
g.students.order_by(lambda s: s.name).limit(3, offset=3)
```

page(*pagenum*, *pagesize=10*)
This query can be used for displaying the second page of group 101 student's list ordered by the `name` attribute:

```
g.students.order_by(Student.name).page(2, pagesize=3)
g.students.order_by(lambda s: s.name).limit(3, offset=3)
```

random(*limit*)
Return a number of random objects from a collection.

```
g = Group[101]
g.students.random(2)
```

remove(*item|iter*)
Remove an item or items from the collection and thus break the relationship between entity instances.

select()
Select objects from a collection. The method names *select()* and *filter()* are synonyms. Example:

```
g = Group[101]
g.students.select(lambda student: student.gpa > 3)
```

Entity options

table
Specify the name of mapped table in the database. See more information in the *Mapping customization* section.

discriminator
Specify the discriminator value for an entity. See more information in the *Entity inheritance* section.

PrimaryKey(*attrs*)
Combine a primary key from multiple attributes. *Link*.

composite_key(*attrs*)
Combine a secondary key from multiple attributes. *Link*.

composite_index(*attrs*)
Combine an index from multiple attributes. *Link*.

Entity hooks

Sometimes you might need to perform an action before or after your entity instance is going to be created, updated or deleted in the database. For this purpose you can use entity hooks.

Here is the list of available hooks:

after_delete()
Called after the entity instance is deleted in the database.

after_insert()
Called after the row is inserted into the database.

after_update()
Called after the instance updated in the database.

before_delete()
Called before deletion the entity instance in the database.

before_insert()
Called only for newly created objects before it is inserted into the database.

before_update()
Called for entity instances before updating the instance in the database.

In order to use a hook, you need to define an entity method with the hook name:

```
class Message(db.Entity):
    title = Required(str)
    content = Required(str)

    def before_insert(self):
        print("Before insert! title=%s" % self.title)
```

Each hook method receives the instance of the object to be modified. You can check how it works in the interactive mode:

```
>>> m = Message(title='First message', content='Hello, world!')
>>> commit()
Before insert! title=First message

INSERT INTO "Message" ("title", "content") VALUES (?, ?)
[u'First message', u'Hello, world!']
```

Entity methods

class Entity

classmethod __getitem__()
Return an entity instance selected by its primary key. Raises the `ObjectNotFound` exception if there is no such object. Example:

```
p = Product[123]
```

For entities with a composite primary key, use a comma between the primary key values:

```
item = OrderItem[123, 456]
```

If object with the specified primary key was already loaded into the *db_session()* cache, Pony returns the object from the cache without sending a query to the database.

delete()
Delete the entity instance. The instance will be marked as deleted and then will be deleted from the database during the *flush()* function, which is issued automatically on committing the current transaction when exiting from the most outer *db_session()* or before sending the next query to the database.

```
Order[123].delete()
```

classmethod describe()
Return a string with the entity declaration.

```
>>> print(OrderItem.describe())

class OrderItem(Entity):
    quantity = Required(int)
    price = Required(Decimal)
    order = Required(Order)
    product = Required(Product)
    PrimaryKey(order, product)
```

classmethod drop_table(*with_all_data=False*)
Drops the table which is associated with the entity in the database. If the table is not empty and `with_all_data=False`, the method raises the `TableIsNotEmpty` exception and doesn't delete anything. Setting the `with_all_data=True` allows you to delete the table even if it is not empty.

If you need to delete an intermediate table created for many-to-many relationship, you have to call the method *select()* of the relationship attribute.

classmethod exists(**args*, ***kwargs*)
Returns `True` if an instance with the specified condition or attribute values exists and `False` otherwise.

```
Product.exists(price=1000)
Product.exists(lambda p: p.price > 1000)
```

flush()
Save the changes made to this object to the database. Usually Pony saves changes automatically and you don't need to call this method yourself. One of the use cases when it might be needed is when you want to get the primary key value of a newly created object which has autoincremented primary key before commit.

classmethod get(**args*, ***kwargs*)
Extract one entity instance from the database.

If the object with the specified parameters exists, then returns the object. Returns `None` if there is no such object. If there are more than one objects with the specified parameters, raises the `MultipleObjectsFoundError: Multiple objects were found. Use select(...) to retrieve them` exception. Examples:

```
Product.get(price=1000)
Product.get(lambda p: p.name.startswith('A'))
```

classmethod get_by_sql(*sql*, *globals=None*, *locals=None*)

Select entity instance by raw SQL.

If you find that you cannot express a query using the standard Pony queries, you always can write your own SQL query and Pony will build an entity instance(s) based on the query results. When Pony gets the result of the SQL query, it analyzes the column names which it receives from the database cursor. If your query uses `SELECT * ...` from the entity table, that would be enough for getting the necessary attribute values for constructing entity instances. You can pass parameters into the query, see *Using the select_by_sql() and get_by_sql() methods* for more information.

classmethod get_for_update(**args*, ***kwargs*, *nowait=False*)

Parameters **nowait** (*bool*) – prevent the operation from waiting for other transactions to commit. If a selected row(s) cannot be locked immediately, the operation reports an error, rather than waiting.

Locks the row in the database using the `SELECT ... FOR UPDATE` SQL query. If `nowait=True`, then the method will throw an exception if this row is already blocked. If `nowait=False`, then it will wait if the row is already blocked.

If you need to use `SELECT ... FOR UPDATE` for multiple rows then you should use the *for_update()* method.

get_pk()

Get the value of the primary key of the object.

```
>>> c = Customer[1]
>>> c.get_pk()
1
```

If the primary key is composite, then this method returns a tuple consisting of primary key column values.

```
>>> oi = OrderItem[1,4]
>>> oi.get_pk()
(1, 4)
```

load(**args*)

Load all lazy and non-lazy attributes, but not collection attributes, which were not retrieved from the database yet. If an attribute was already loaded, it won't be loaded again. You can specify the list of the attributes which need to be loaded, or it's names. In this case Pony will load only them:

```
obj.load(Person.biography, Person.some_other_field)
obj.load('biography', 'some_other_field')
```

classmethod select(*lambda*)

Select objects from the database in accordance with the condition specified in lambda, or all objects if lambda function is not specified.

The `select()` method returns an instance of the *Query* class. Entity instances will be retrieved from the database once you start iterating over the `Query` object.

This query example returns all products with the price greater than 100 and which were ordered more than once:

```
Product.select(lambda p: p.price > 100 and count(p.order_items) > 1)[:]
```

classmethod **select_by_sql**(*sql*, *globals=None*, *locals=None*)
Select entity instances by raw SQL. See *Using the select_by_sql() and get_by_sql() methods* for more information.

classmethod **select_random**(*limit*)
Select `limit` random objects. This method uses the algorithm that can be much more effective than using `ORDER BY RANDOM()` SQL construct. The method uses the following algorithm:

1. Determine max id from the table.
2. Generate random ids in the range (0, max_id]
3. Retrieve objects by those random ids. If an object with generated id does not exist (e.g. it was deleted), then select another random id and retry.

Repeat the steps 2-3 as many times as necessary to retrieve the specified amount of objects.

This algorithm doesn't affect performance even when working with a large number of table rows. However this method also has some limitations:

- The primary key must be a sequential id of an integer type.
- The number of "gaps" between existing ids (the count of deleted objects) should be relatively small.

The `select_random()` method can be used if your query does not have any criteria to select specific objects. If such criteria is necessary, then you can use the *`Query.random()`* method.

set(***kwargs*)
Assign new values to several object attributes at once:

```
Customer[123].set(email='new@example.com', address='New address')
```

This method also can be convenient when you want to assign new values from a dictionary:

```
d = {'email': 'new@example.com', 'address': 'New address'}
Customer[123].set(**d)
```

to_dict(*only=None*, *exclude=None*, *with_collections=False*, *with_lazy=False*, *related_objects=False*)
Return a dictionary with attribute names and its values. This method can be used when you need to serialize an object to JSON or other format.

By default this method doesn't include collections (to-many relationships) and lazy attributes. If an attribute's values is an entity instance then only the primary key of this object will be added to the dictionary.

Parameters

- **only** (*list | str*) – use this parameter if you want to get only the specified attributes. This argument can be used as a first positional argument. You can specify a list of attribute names `obj.to_dict(['id', 'name'])`, a string separated by spaces: `obj.to_dict('id name')`, or a string separated by spaces with commas: `obj.to_dict('id, name')`.
- **exclude** (*list | str*) – this parameter allows you to exclude specified attributes. Attribute names can be specified the same way as for the `only` parameter.
- **related_objects** (*bool*) – by default, all related objects represented as a primary key. If `related_objects=True`, then objects which have relationships with the current object will be added to the resulting dict as objects, not their primary keys. It can be useful if you want to walk the related objects and call the `to_dict()` method recursively.

- **with_collections** (*bool*) – by default, the resulting dictionary will not contain collections (to-many relationships). If you set this parameter to `True`, then the relationships to-many will be represented as lists. If `related_objects=False` (which is by default), then those lists will consist of primary keys of related instances. If `related_objects=True` then to-many collections will be represented as lists of objects.
- **with_lazy** (*bool*) – if `True`, then lazy attributes (such as BLOBs or attributes which are declared with `lazy=True`) will be included to the resulting dict.
- **related_objects** – By default all related objects are represented as a list with their primary keys only. If you want to see the related objects instances, you can specify `related_objects=True`.

For illustrating the usage of this method we will use the eStore example which comes with Pony distribution. Let's get a customer object with the id=1 and convert it to a dictionary:

```
>>> from pony.orm.examples.estore import *
>>> c1 = Customer[1]
>>> c1.to_dict()

{'address': u'address 1',
'country': u'USA',
'email': u'john@example.com',
'id': 1,
'name': u'John Smith',
'password': u'***'}
```

If we don't want to serialize the password attribute, we can exclude it this way:

```
>>> c1.to_dict(exclude='password')

{'address': u'address 1',
'country': u'USA',
'email': u'john@example.com',
'id': 1,
'name': u'John Smith'}
```

If you want to exclude more than one attribute, you can specify them as a list: `exclude=['id', 'password']` or as a string: `exclude='id, password'` which is the same as `exclude='id password'`.

Also you can specify only the attributes, which you want to serialize using the parameter `only`:

```
>>> c1.to_dict(only=['id', 'name'])

{'id': 1, 'name': u'John Smith'}

>>> c1.to_dict('name email') # 'only' parameter as a positional argument

{'email': u'john@example.com', 'name': u'John Smith'}
```

By default the collections are not included to the resulting dict. If you want to include them, you can specify `with_collections=True`. Also you can specify the collection attribute in the `only` parameter:

```
>>> c1.to_dict(with_collections=True)

{'address': u'address 1',
'cart_items': [1, 2],
```

```
'country': u'USA',
'email': u'john@example.com',
'id': 1,
'name': u'John Smith',
'orders': [1, 2],
'password': u'***'}
```

By default all related objects (cart_items, orders) are represented as a list with their primary keys. If you want to see the related objects instances, you can specify `related_objects=True`:

```
>>> c1.to_dict(with_collections=True, related_objects=True)

{'address': u'address 1',
'cart_items': [CartItem[1], CartItem[2]],
'country': u'USA',
'email': u'john@example.com',
'id': 1,
'name': u'John Smith',
'orders': [Order[1], Order[2]],
'password': u'***'}
```

Queries and functions

Below is the list of upper level functions defined in Pony:

avg(*gen*)

Return the average value for all selected attributes.

Parameters **gen** (*generator*) – Python generator expression

Return type numeric

```
avg(o.total_price for o in Order)
```

The equivalent query can be generated using the *avg()* method.

concat(**args*)

Parameters **args** (*list*) – list of arguments

Concatenates arguments into one string.

```
select(concat(p.first_name, ' ', p.last_name) for p in Person)
```

commit()

Save all changes which were made within the current *db_session()* using the *flush()* function and commits the transaction to the database. This top level *commit()* function calls the *commit()* method of each database object which was used in current transaction.

count(*gen*)

Return the number of objects that match the query condition.

Parameters **gen** (*generator*) – Python generator expression

Return type numeric

```
count(c for c in Customer if len(c.orders) > 2)
```

This query will be translated to the following SQL:

```
SELECT COUNT(*)
FROM "Customer" "c"
LEFT JOIN "Order" "order-1"
  ON "c"."id" = "order-1"."customer"
GROUP BY "c"."id"
HAVING COUNT(DISTINCT "order-1"."id") > 2
```

The equivalent query can be generated using the *count()* method.

delete(*gen*)

Delete objects from the database. Pony loads objects into the memory and will delete them one by one. If you have *before_delete()* or *after_delete()* defined, Pony will call each of them.

Parameters **gen** (*generator*) – Python generator expression

```
delete(o for o in Order if o.status == 'CANCELLED')
```

If you need to delete objects without loading them into memory, you should use the *delete()* method with the parameter `bulk=True`. In this case no hooks will be called, even if they are defined for the entity.

desc(*attr*)

This function is used inside *order_by()* for ordering in descending order.

Parameters **attr** (*attribute*) – Entity attribute

```
select(o for o in Order).order_by(desc(Order.date_shipped))
```

The same example, using `lambda`:

```
select(o for o in Order).order_by(lambda o: desc(o.date_shipped))
```

distinct(*gen*)

When you need to force DISTINCT in a query, it can be done using the `distinct()` function. But usually this is not necessary, because Pony adds DISTINCT keyword automatically in an intelligent way. See more information about it in the TODO chapter.

Parameters **gen** (*generator*) – Python generator expression

```
distinct(o.date_shipped for o in Order)
```

Another usage of the *distinct()* function is with the *sum()* aggregate function - you can write:

```
select(sum(distinct(x.val)) for x in X)
```

to generate the following SQL:

```
SELECT SUM(DISTINCT x.val)
FROM X x
```

but it is rarely used in practice.

exists(*gen*, *globals=None*, *locals=None*)

Returns *True* if at least one instance with the specified condition exists and *False* otherwise.

Parameters

- **gen** (*generator*) – Python generator expression.
- **globals** (*dict*) –

- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Return type bool

```
exists(o for o in Order if o.date_delivered is None)
```

flush()

Save all changes from the *db_session()* cache to the databases, without committing them. It makes the updates made in the *db_session()* cache visible to all database queries which belong to the current transaction.

Usually Pony saves data from the database session cache automatically and you don't need to call this function yourself. One of the use cases when it might be needed is when you want to get the primary keys values of newly created objects which has autoincremented primary key before commit.

This top level `flush()` function calls the *flush()* method of each database object which was used in current transaction.

This function is called automatically before executing the following functions: *commit()*, *get()*, *exists()*, *select()*.

get(*gen*, *globals=None*, *locals=None*)

Extracts one entity instance from the database.

Parameters

- **gen** (*generator*) – Python generator expression.
- **globals** (*dict*) –
- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Returns the object if an object with the specified parameters exists, or `None` if there is no such object.

If there are more than one objects with the specified parameters, the function raises the `MultipleObjectsFoundError: Multiple objects were found. Use select(...) to retrieve them` exception.

```
get(o for o in Order if o.id == 123)
```

The equivalent query can be generated using the *get()* method.

getattr(*object*, *name*[, *default*])

This is a standard Python built-in function, that can be used for getting the attribute value inside the query.

Example:

```
attr_name = 'name'
param_value = 'John'
select(c for c in Customer if getattr(c, attr_name) == param_value)
```

JOIN(**args*)

Used for query optimization in cases when Pony doesn't provide this optimization automatically. Serves as a hint saying Pony that we want to use SQL JOIN, instead of generating a subquery inside the SQL query.

```
select(g for g in Group if max(g.students.gpa) < 4)

select(g for g in Group if JOIN(max(g.students.gpa) < 4))
```

left_join(*gen*, *globals=None*, *locals=None*)

The results of a left join always contain the result from the 'left' table, even if the join condition doesn't find any matching record in the 'right' table.

Parameters

- **gen** (*generator*) – Python generator expression.
- **globals** (*dict*) –
- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Let's say we need to calculate the amount of orders for each customer. Let's use the example which comes with Pony distribution and write the following query:

```
from pony.orm.examples.estore import *
populate_database()

select((c, count(o)) for c in Customer for o in c.orders)[:]
```

It will be translated to the following SQL:

```
SELECT "c"."id", COUNT(DISTINCT "o"."id")
FROM "Customer" "c", "Order" "o"
WHERE "c"."id" = "o"."customer"
GROUP BY "c"."id"
```

And return the following result:

```
[(Customer[1], 2), (Customer[2], 1), (Customer[3], 1), (Customer[4], 1)]
```

But if there are customers that have no orders, they will not be selected by this query, because the condition `WHERE "c"."id" = "o"."customer"` doesn't find any matching record in the Order table. In order to get the list of all customers, we should use the `left_join()` function:

```
left_join((c, count(o)) for c in Customer for o in c.orders)[:]
```

```
SELECT "c"."id", COUNT(DISTINCT "o"."id")
FROM "Customer" "c"
LEFT JOIN "Order" "o"
  ON "c"."id" = "o"."customer"
GROUP BY "c"."id"
```

Now we will get the list of all customers with the number of order equal to zero for customers which have no orders:

```
[(Customer[1], 2), (Customer[2], 1), (Customer[3], 1), (Customer[4], 1),␣
↪(Customer[5], 0)]
```

We should mention that in most cases Pony can understand where LEFT JOIN is needed. For example, the same query can be written this way:

```
select((c, count(c.orders)) for c in Customer)[:]
```

```
SELECT "c"."id", COUNT(DISTINCT "order-1"."id")
FROM "Customer" "c"
LEFT JOIN "Order" "order-1"
```

```
  ON "c"."id" = "order-1"."customer"
GROUP BY "c"."id"
```

len(*arg*)

Return the number of objects in the collection. Can be used only within the query, similar to *count()*.

Parameters **arg** (*generator*) – a collection

Return type numeric

```
Customer.select(lambda c: len(c.orders) > 2)
```

max(*gen*)

Return the maximum value from the database. The query should return a single attribute.

Parameters **gen** (*generator*) – Python generator expression.

```
max(o.date_shipped for o in Order)
```

The equivalent query can be generated using the *max()* method.

min(**args*, ***kwargs*)

Return the minimum value from the database. The query should return a single attribute.

Parameters **gen** (*generator*) – Python generator expression.

```
min(p.price for p in Product)
```

The equivalent query can be generated using the *min()* method.

random()

Returns a random value from 0 to 1. This functions, when encountered inside a query will be translated into RANDOM SQL query.

Example:

```
select(s.gpa for s in Student if s.gpa > random() * 5)
```

```
SELECT DISTINCT "s"."gpa"
FROM "student" "s"
WHERE "s"."gpa" > (random() * 5)
```

raw_sql(*sql*, *result_type=None*)

This function encapsulates a part of a query expressed in a raw SQL format. If the `result_type` is specified, Pony converts the result of raw SQL fragment to the specified format.

Parameters

- **sql** (*str*) – SQL statement text.
- **result_type** (*type*) – the type of the SQL statement result.

```
>>> q = Person.select(lambda x: raw_sql('abs("x"."age")') > 25)
>>> print(q.get_sql())
```

```
SELECT "x"."id", "x"."name", "x"."age", "x"."dob"
FROM "Person" "x"
WHERE abs("x"."age") > 25
```

```
x = 10
y = 15
select(p for p in Person if raw_sql('p.age > $(x + y)'))

names = select(raw_sql('UPPER(p.name)') for p in Person)[:]
print(names)

['JOHN', 'MIKE', 'MARY']
```

See more examples *here*.

rollback()
Roll back the current transaction.

This top level `rollback()` function calls the *rollback()* method of each database object which was used in current transaction.

select(*gen*)
Translates the generator expression into SQL query and returns an instance of the *Query* class.

Parameters

- **gen** (*generator*) – Python generator expression.
- **globals** (*dict*) –
- **locals** (*dict*) – optional parameters which can contain dicts with variables and its values, used within the query.

Return type *Query* or list

You can iterate over the result:

```
for p in select(p for p in Product):
    print p.name, p.price
```

If you need to get a list of objects you can get a full slice of the result:

```
prod_list = select(p for p in Product)[:]
```

The `select()` function can also return a list of single attributes or a list of tuples:

```
select(p.name for p in Product)

select((p1, p2) for p1 in Product
                for p2 in Product if p1.name == p2.name and p1 != p2)

select((p.name, count(p.orders)) for p in Product)
```

You can apply any *Query* method to the result, e.g. *order_by()* or *count()*.

If you want to run a query over a relationship attribute, you can use the *select()* method of the relationship attribute.

show()
Prints out the entity definition or the value of attributes for an entity instance in the interactive mode.

Parameters **value** – entity class or entity instance

```
>>> show(Person)
class Person(Entity):
    id = PrimaryKey(int, auto=True)
```

```
    name = Required(str)
    age = Required(int)
    cars = Set(Car)


>>> show(mary)
instance of Person
id|name|age
--+----+---
2 |Mary|22
```

sql_debug(*value*)

Prints SQL statements being sent to the database to the console or to a log file.

Parameters **value** (*bool*) – sets debugging on/off

By default Pony sends debug information to stdout. If you have the standard Python logging configured, Pony will use it instead. Here is how you can store debug information in a file:

```
import logging
logging.basicConfig(filename='pony.log', level=logging.INFO)
```

Note, that we had to specify the `level=logging.INFO` because the default standard logging level is `WARNING` and Pony uses the `INFO` level for its messages by default. Pony uses two loggers: `pony.orm.sql` for SQL statements that it sends to the database and `pony.orm` for all other messages.

sum(*gen*)

Return the sum of all values selected from the database.

Parameters **gen** (*generator*) – Python generator expression

Return type numeric

Returns a number. If the query returns no items, the `sum()` method returns 0.

```
sum(o.total_price for o in Order)
```

The equivalent query can be generated using the *sum()* method.

Query object

The generator expression and lambda queries return an instance of the `Query` class. Below is the list of methods that you can apply to it.

class Query

[start:end]
[index]

Limit the number of instances to be selected from the database. In the example below we select the first ten instances:

```
# generator expression query
select(c for c in Customer)[:10]

# lambda function query
Customer.select()[:10]
```

Generates the following SQL:

```
SELECT "c"."id", "c"."email", "c"."password", "c"."name", "c"."country", "c".
↪"address"
FROM "Customer" "c"
LIMIT 10
```

If we need to select instances with offset, we should use `start` and `end` values:

```
select(c for c in Customer).order_by(Customer.name)[20:30]
```

It generates the following SQL:

```
SELECT "c"."id", "c"."email", "c"."password", "c"."name", "c"."country", "c".
↪"address"
FROM "Customer" "c"
ORDER BY "c"."name"
LIMIT 10 OFFSET 20
```

Also you can use the *limit()* or *page()* methods for the same purpose.

__len__()
Return the number of objects selected from the database.

```
len(select(c for c in Customer))
```

avg()
Return the average value for all selected attributes:

```
select(o.total_price for o in Order).avg()
```

The function *avg()* does the same thing.

count()

Return the number of objects that match the query condition:

```
select(c for c in Customer if len(c.orders) > 2).count()
```

The function *count()* does the same thing.

delete(*bulk=None*)
Delete instances selected by a query. When `bulk=False` Pony loads each instance into memory and call the *Entity.delete()* method on each instance (calling *before_delete()* and *after_delete()* hooks if they are defined). If `bulk=True` Pony doesn't load instances, it just generates the SQL DELETE statement which deletes objects in the database.

Note: Be careful with the bulk delete:

- *before_delete()* and *after_delete()* hooks will not be called on deleted objects.
- If an object was loaded into memory, it will not be removed from the *db_session()* cache on bulk delete.

---

distinct()
Force DISTINCT in a query:

```
select(c.name for c in Customer).distinct()
```

But usually this is not necessary, because Pony adds DISTINCT keyword automatically in an intelligent way. See more information about it in the *Automatic DISTINCT* section.

The function *distinct()* does the same thing.

exists()

Returns `True` if at least one instance with the specified condition exists and `False` otherwise:

```
select(c for c in Customer if len(c.cart_items) > 10).exists()
```

This query generates the following SQL:

```
SELECT "c"."id"
FROM "Customer" "c"
  LEFT JOIN "CartItem" "cartitem-1"
    ON "c"."id" = "cartitem-1"."customer"
GROUP BY "c"."id"
HAVING COUNT(DISTINCT "cartitem-1"."id") > 20
LIMIT 1
```

filter(*lambda*, *globals=None*, *locals=None*)
filter(*str*)
filter(***kwargs*)

Filter the result of a query. The conditions which are passed as parameters to the `filter()` method will be translated into the WHERE section of the resulting SQL query.

Before Pony ORM release 0.5 the `filter()` method affected the underlying query updating the query in-place, but since the release 0.5 it creates and returns a new `Query` object with the applied conditions.

The number of `filter()` arguments should correspond to the query result. The `filter()` method can receive a lambda expression with a condition:

```
q = select(p for p in Product)
q2 = q.filter(lambda x: x.price > 100)

q = select((p.name, p.price) for p in Product)
q2 = q.filter(lambda n, p: n.name.startswith("A") and p > 100)
```

Also the `filter()` method can receive a text string where you can specify just the expression:

```
q = select(p for p in Product)
x = 100
q2 = q.filter("p.price > x")
```

Another way to filter the query result is to pass parameters in the form of named arguments:

```
q = select(p for p in Product)
q2 = q.filter(price=100, name="iPod")
```

first()

Return the first element from the selected results or `None` if no objects were found:

```
select(p for p in Product if p.price > 100).first()
```

for_update(*nowait=False*)

Parameters **nowait** (*bool*) – prevent the operation from waiting for other transactions to commit. If a selected row(s) cannot be locked immediately, the operation reports an error, rather than waiting.

Sometimes there is a need to lock objects in the database in order to prevent other transactions from modifying the same instances simultaneously. Within the database such lock should be done using the SELECT FOR UPDATE query. In order to generate such a lock using Pony you can call the `for_update` method:

```
select(p for p in Product if p.picture is None).for_update()
```

This query selects all instances of Product without a picture and locks the corresponding rows in the database. The lock will be released upon commit or rollback of current transaction.

get()

Extract one entity instance from the database. The function returns the object if an object with the specified parameters exists, or `None` if there is no such object. If there are more than one objects with the specified parameters, raises the `MultipleObjectsFoundError: Multiple objects were found. Use select(...) to retrieve them` exception. Example:

```
select(o for o in Order if o.id == 123).get()
```

The function *get()* does the same thing.

get_sql()

Return SQL statement as a string:

```
sql = select(c for c in Category if c.name.startswith('a')).get_sql()
print(sql)
```

```
SELECT "c"."id", "c"."name"
FROM "category" "c"
WHERE "c"."name" LIKE 'a%%'
```

limit(*limit*, *offset=None*)

Limit the number of instances to be selected from the database.

```
select(c for c in Customer).order_by(Customer.name)[20:30]
```

Also you can use the `[start:end]()` or *page()* methods for the same purpose.

max()

Return the maximum value from the database. The query should return a single attribute:

```
select(o.date_shipped for o in Order).max()
```

The function *max()* does the same thing.

min()

Return the minimum value from the database. The query should return a single attribute:

```
select(p.price for p in Product).min()
```

The function *min()* does the same thing.

order_by(*attr1*[, *attr2*, ...])
order_by(*pos1*[, *pos2*, ...])
order_by(*lambda[, globals[, locals]]*)

order_by(*str*)

Order the results of a query. There are several options available:

•Using entity attributes

```
select(o for o in Order).order_by(Order.customer, Order.date_created)
```

For ordering in descending order, use the function *desc()*:

```
select(o for o in Order).order_by(desc(Order.date_created))
```

•Using position of query result variables

```
select((o.customer.name, o.total_price) for o in Order).order_by(-2, 1)
```

The position numbers start with 1. Minus means sorting in the descending order. In this example we sort the result by the total price in descending order and by the customer name in ascending order.

•Using lambda

```
select(o for o in Order).order_by(lambda o: (o.customer.name, desc(o.date_
↪shipped)))
```

If the lambda has a parameter (`o` in our example) then `o` represents the result of the `select` and will be applied to it. If you specify the lambda without a parameter, then inside lambda you have access to all names defined inside the query:

```
select(o.total_price for o in Order).order_by(lambda: o.customer.id)
```

•Using a string

This approach is similar to the previous one, but you specify the body of a lambda as a string:

```
select(o for o in Order).order_by("o.customer.name, desc(o.date_shipped)")
```

page(*pagenum*, *pagesize=10*)

Pagination is used when you need to display results of a query divided into multiple pages. The page numbering starts with page 1. This method returns a slice [start:end] where `start = (pagenum - 1) * pagesize`, `end = pagenum * pagesize`.

prefetch(**args*)

Allows specifying which related objects or attributes should be loaded from the database along with the query result.

Usually there is no need to prefetch related objects. When you work with the query result within the `@db_session`, Pony gets all related objects once you need them. Pony uses the most effective way for loading related objects from the database, avoiding the N+1 Query problem.

So, if you use Flask, the recommended approach is to use the `@db_session` decorator at the top level, at the same place where you put the Flask's `app.route` decorator:

```
@app.route('/index')
@db_session
def index():
    ...
    objects = select(...)
```

```
    ...
    return render_template('template.html', objects=objects)
```

Or, even better, wrapping the wsgi application with the *db_session()* decorator:

```
app.wsgi_app = db_session(app.wsgi_app)
```

If for some reason you need to pass the selected instances along with related objects outside of the *db_session()*, then you can use this method. Otherwise, if you'll try to access the related objects outside of the *db_session()*, you might get the `DatabaseSessionIsOver` exception, e.g. `DatabaseSessionIsOver: Cannot load attribute Customer[3].name: the database session is over`

More information regarding working with the *db_session()* can be found *here*.

You can specify entities and/or attributes as parameters. When you specify an entity, then all "to-one" and non-lazy attributes of corresponding related objects will be prefetched. The "to-many" attributes of an entity are prefetched only when specified explicitly.

If you specify an attribute, then only this specific attribute will be prefetched. You can specify attribute chains, e.g. `order.customer.address`. The prefetching works recursively - it applies the specified parameters to each selected object.

Examples:

```
from pony.orm.examples.presentation import *
```

Loading Student objects only, without prefetching:

```
students = select(s for s in Student)[:]
```

Loading students along with groups and departments:

```
students = select(s for s in Student).prefetch(Group, Department)[:]

for s in students: # no additional query to the DB will be sent
    print s.name, s.group.major, s.group.dept.name
```

The same as above, but specifying attributes instead of entities:

```
students = select(s for s in Student).prefetch(Student.group, Group.dept)[:]

for s in students: # no additional query to the DB will be sent
    print s.name, s.group.major, s.group.dept.name
```

Loading students and related courses ("many-to-many" relationship):

```
students = select(s for s in Student).prefetch(Student.courses)

for s in students:
    print s.name
    for c in s.courses: # no additional query to the DB will be sent
        print c.name
```

random(*limit*)

Select `limit` random objects from the database. This method will be translated using the `ORDER BY RANDOM()` SQL expression. The entity class method *select_random()* provides better performance, although doesn't allow to specify query conditions.

For example, select ten random persons older than 20 years old:

```
select(p for p in Person if p.age > 20).random()[:10]
```

show(*width=None*)

Prints the results of a query to the console. The result is formatted in the form of a table. This method doesn't display "to-many" attributes because it would require additional query to the database and could be bulky. But if an instance has a "to-one" relationship, then it will be displayed.

```
>>> select(p for p in Person).order_by(Person.name)[:2].show()

SELECT "p"."id", "p"."name", "p"."age"
FROM "Person" "p"
ORDER BY "p"."name"
LIMIT 2

id|name|age
--+----+---
3 |Bob |30

>>> Car.select().show()
id|make  |model   |owner
--+------+--------+---------
1 |Toyota|Prius   |Person[2]
2 |Ford  |Explorer|Person[3]
```

sum()

Return the sum of all selected items. Can be applied to the queries which return a single numeric expression only.

```
select(o.total_price for o in Order).sum()
```

If the query returns no items, the query result will be 0.

to_json(*include=()*, *exclude=()*, *converter=None*, *with_schema=True*, *schema_hash=None*)

without_distinct()

By default Pony tries to avoid duplicates in the query result and intellectually adds the `DISTINCT` SQL keyword to a query where it thinks it necessary. If you don't want Pony to add `DISTINCT` and get possible duplicates, you can use this method. This method returns a new instance of the Query object, so you can chain it with other query methods:

```
select(p.name for p in Person).without_distinct().order_by(Person.name)
```

Before Pony Release 0.6 the method `without_distinct()` returned query result and not a new query instance.

Statistics - QueryStat

The `Database` object has a thread-local property *`local_stats`* which contains query execution statistics. The property value is a dict, where keys are SQL queries and values are instances of the `QueryStat` class. A `QueryStat` object has the following attributes:

class QueryStat

sql
The text of SQL query

db_count
The number of times this query was sent to the database

cache_count
The number of times the query result was taken directly from the *db_session()* cache (for cases when a query was called repeatedly inside the same *db_session()*)

min_time
The minimum time required for database to execute the query

max_time
The maximum time required for database to execute the query

avg_time
The average time required for database to execute the query

sum_time
Total time spent (is equal to *avg_time* * *db_count*)

Pony keeps all statistics separately for each thread. If you want to see the aggregated statistics for all threads then you need to call the *merge_local_stats()* method. See also: *local_stats()*, *global_stats()*, .

Example:

```
query_stats = sorted(db.local_stats.values(),
        reverse=True, key=attrgetter('sum_time'))
for qs in query_stats:
    print(qs.sum_time, qs.db_count, qs.sql)
```

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W