A common programming task is to abstract. Abstract the inner logic until complex tasks can be encapsulated into simple functions. For a very long time, the only way to access was to write out queries. You would find code like this sprinkled everywhere,

```
INSERT INTO categories VALUES(1, 'technology')
```

However this is not readable or maintainable and terribly error prone. The natural step was to abstract this out. This resulted in ORMs (Object Relational Mapping) being built. ORM is a programming technique for converting data between incompatible type systems in object-oriented programming languages. With ORMs you'll be able to map class object relationships as table column relationships in SQL databases.

> SQLAlchemy is an open source SQL toolkit and object-relational mapper (ORM) for the Python programming language released under the MIT License.

SQLAlchemy's philosophy is that SQL databases behave less and less like object collections the more size and performance start to matter, while object collections behave less and less like tables and rows the more abstraction starts to matter. For this reason it has adopted the data mapper pattern (like Hibernate for Java) rather than the active record pattern used by a number of other object-relational mappers.

However, I do not prefer the data mapper approach. I feel it requires far too much boilerplate to be written before you get to use it as a full blown ORM. I think others have also felt this way, which is why SQLAlchemy also has a declarative approach that tries to map the active record pattern. That's what we will be exploring in this article.

SQLAlchemy Essentials

1. Engine

> The Engine is the starting point for any SQLAlchemy application. It's "home base" for the actual database and its DBAPI, delivered to the SQLAlchemy application through a connection pool and a Dialect, which describes how to talk to a specific kind of database/DBAPI combination.

```
engine = create_engine('mysql://root:root@127.0.0.1:3306', convert_unicode=True)
```

1. Declarative Base

You will be required to inherit all your custom classes from this base class. Declarative Base is used to connect the class and object attributes to the database's tables and column relationships.

```
Base = declarative_base()
```

1. Session

The session establishes all conversations with the database and represents a "staging zone" for all the objects loaded into the database session object. Any change made against the objects in the session won't be persisted into the database until you call session.commit(). If you're not happy about the changes, you can revert all of them back to the last commit by calling session.rollback()

```
db_session = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

1. Binding the session to the engine At the end, you will need to create all the tables and bind the current session to the database engine of your choice. This will ensure that all object attributes are reflected in the database tables and columns.

```
Base.metadata.create_all(engine)
Base.metadata.bind = engine
```

Building with SQLAlchemy

Let's create a simple Python package that creates notes and categorizes them. We'll use SQLAlchemy and MySQL for storing the notes and categories.

I always like starting out with a `conf` file that initializes the engine, session and base. It also has important functions to create, initialize, drop and save to the database.

```
import os
from sqlalchemy import create_engine
from sqlalchemy.orm import relationship, sessionmaker
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, ForeignKey, Integer, String, Text

engine = create_engine('mysql://root:root@127.0.0.1:3306', convert_unicode=True)

Base = declarative_base()

db_session = sessionmaker(autocommit=False, autoflush=False, bind=engine)
session = db_session()

def create_db():
  engine.execute("CREATE DATABASE IF NOT EXISTS {0}".format(db_name))
  engine.execute("USE {0}".format(db_name))

def init_db():
  create_db()

  # Creates all the tables in the database
```

```
    # Will skip already created tables
    Base.metadata.create_all(engine)
    Base.metadata.bind = engine

  def drop_db():
    engine.execute("DROP DATABASE IF EXISTS {0}".format(db_name))

  def save_to_db(record):
    try:
      session.add(record)
      session.commit()
    except Exception as e:
      print e
```

Let's create a Category class and link it to the `categories` MySQL table by inheriting from `Base`. Since the `conf` file has already initialized the Base class, we just have to import it.

```
import conf

class Category(conf.Base):
  __tablename__ = 'categories'

  id = conf.Column(conf.Integer, primary_key=True)
  name = conf.Column(conf.String(10), nullable=False, unique=True)

  @classmethod
  def search_by_name(cls, name):
    return conf.session.query(Category).filter(Category.name == name)

  @classmethod
  def find_by_name(cls, name):
    return conf.session.query(Category).filter(Category.name == name).one()

  def save(self):
    conf.save_to_db(self)
```

The `__tablename__` attribute defines the name of the MySQL table that the `Category` class will communicate to. `id` and `name` are the columns of the `categories` MySQL table and the attributes of the `Category` class.

If you want to query for all records that have a certain name, just abstract `conf.session` to Category class and use the `filter` function.

```
conf.session.query(Category).filter(Category.name == name)
```

If you want to limit the query to just one,

```
conf.session.query(Category).filter(Category.name == name).one()
```

Relationships in SQLAlchemy

In almost every MySQL database, relationships are how tables talk to each other. It can be a one-to-many, one-to-one, many-to-many or many-to-one.

We wanted to categorize notes. We have a Category class, we now need to create a Note class that can communicate with Category.

```
import conf
import category

class Note(conf.Base):
    __tablename__ = 'notes'

    id = conf.Column(conf.Integer, primary_key=True)
    text = conf.Column(conf.Text, nullable=False)
    category_id = conf.Column(
                        conf.Integer,
                        conf.ForeignKey('categories.id'),
                        nullable=False)
    category = conf.relationship(category.Category)

    def save(self):
        conf.save_to_db(self)
```

> ProTip: Make sure to 'import category' as a file not as a class (not 'from category import Category'). Else you'll run into dependency issues during runtime.

Each note has a `category_id` column that is used to map the category this note belongs to. This defines the database relationship.

How do the classes communicate? Just import category here and use the `relationship` method to define it.

```
category = conf.relationship(category.Category)
```

Creating a record with SQLAlchemy

Great. We've added the classes, how do we create a record and commit it to database?

Let's start easy and create a Category object.

```
c = Category(name='tech')
```

```
c.save()
```

> ProTip: If you try creating a Category called 'tech' again, you'll run into an error. Why? We've defined a unique constraint on the 'name' attribute.

Now how do we create a note that belongs to this category?

```
n = Note(text="sqlalchemy is cool", category=c)
n.save()
```

That's it!

Scoping database relationships in instance methods

Wouldn't it be cool to query all notes belonging to a category? So that you can just call,

```
c.notes()
```

Just create a `notes()` instance method to your Category class and add this,

```
import note

def notes(self):
  return conf.session.query(note.Note).filter(note.Note.category_id == self.id)
```

Where's the code?

You can find all of the code I used in this article on my Github repository.