Corral Documentation

Release 0.3

Juan B Cabral

Sep 03, 2018

Contents

Trustworthy and Fully Functional Data Intensive Parallel Pipelines

Corral will solve your pipeline needs by merging a database full connection interface with a MVC model, by making you able of editing your custom schemas and adding the possibility of writting specific processing steps following a intuitive data handling model.

CHAPTER 1

Help & discussion mailing list

Our Google Groups mailing list is here.

CHAPTER 2

License

Corral is under The 3-Clause BSD License

This license allows unlimited redistribution for any purpose as long as its copyright notices and the license's disclaimers of warranty are maintained.

CHAPTER 3

Citation

If you are using Corral in your research, please cite:

Note: Juan B. Cabral, Bruno Sánchez, Martín Beroiz, Mariano Domínguez, Marcelo Lares, Sebastián Gurovich: "Corral Framework: Trustworthy and Fully Functional Data Intensive Parallel Astronomical Pipelines", 2017; https://doi.org/10.1016/j.ascom.2017.07.003.

Full Paper: https://arxiv.org/abs/1701.05566

CHAPTER 4

Contents:

4.1 Quick install guide

Before you can use Corral, you'll need to get it installed. We have a complete installation guide that covers all the possibilities; this guide will guide you to a simple, minimal installation that'll work while you walk through the introduction.

4.1.1 Install Python

Being a Python framework, Corral requires Python. Python includes a lightweight database called SQLite so you won't need to set up a database just yet.

Get the latest version of Python at https://www.python.org/download/ or with your operating system's package manager.

You can verify that Python is installed by typing `python` from your shell; you should see something like:

```
Python 3.4.x
[GCC 4.x] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

4.1.2 Get your database running

If you plan to use Corral's database API functionality, you'll need to make sure a database server is running. Corral supports all the database servers provided by SQLAlchemy

If you are developing a simple project or something you don't plan to deploy in a production environment, SQLite is generally the simplest option as it doesn't require running a separate server. However, SQLite has many differences from other databases, so if you are working on something substantial, it's recommended to develop with the same database as you plan on using in production.

In addition to a database backend, you'll need to make sure your SqlAlchemy database bindings are installed.

4.1.3 Remove any old versions of Corral

If you are upgrading your installation of Corral from a previous version, you will need to uninstall the old Corral version before installing the new version.

If you installed Corral using pip or `easy_install` previously, installing with pip or `easy_install` again will automatically take care of the old version, so you don't need to do it yourself.

If you previously installed Corral using `python setup.py install`, uninstalling is as simple as deleting the `corral` directory from your Python `site-packages`. To find the directory you need to remove, you can run the following at your shell prompt (not the interactive Python prompt):

```
$ python -c "import corral; print(corral.__path__)"
```

4.1.4 Install Corral

Installation instructions are slightly different depending on whether you're installing a distribution-specific package, downloading the latest official release, or fetching the latest development version.

It's easy, no matter which way you choose.

Installing an official release with `pip`

This is the recommended way to install Corral.

1. Install pip. The easiest is to use the standalone pip installer. If your distribution already has `pip` installed, you might need to update it if it's outdated. If it's outdated, you'll know because installation won't work. If you're using an old version of setuptools, you might see some **harmless SyntaxErrors** also.
2. Take a look at virtualenv and virtualenvwrapper. These tools provide isolated Python environments, which are more practical than installing packages systemwide. They also allow installing packages without administrator privileges.
3. After you've created and activated a virtual environment, enter the command `pip install -U corral-pipeline` at the shell prompt.

Installing the development version

If you'd like to be able to update your corral code occasionally with the latest bug fixes and improvements, follow these instructions:

1. Make sure that you have Git installed and that you can run its commands from a shell. (Enter `git help` at a shell prompt to test this.)
2. Check out Corral's main development branch like so:

   ```
   $ git clone git@github.com:toros-astro/corral.git
   ```

   This will create a directory `corral` in your current directory.
3. Make sure that the Python interpreter can load Corral's code. The most convenient way to do this is to use virtualenv, virtualenvwrapper, and pip.
4. After setting up and activating the virtualenv, run the following command:

   ```
   $ pip install -e corral/
   ```

This will make Corral's code importable, and will also make the `corral` utility command available. In other words, you're all set!

When you want to update your copy of the Corral source code, just run the command `git pull` from within the `corral` directory. When you do this, Git will automatically download any changes.

4.2 Tutorial

This section contains a step-by-step by example tutorial to create your own data reduce pipeline with Corral

Contents:

4.2.1 Tutorial - Part #1 - Creating An Empty Project

Let's learn by example.

Throughout this tutorial, we'll walk you through the creation of a basic application pipeline.

We'll assume you have *Corral installed* already. You can tell Corral is installed and which version by running the following command:

```
$ python -c "import corral; print(corral.VERSION)"
```

If Corral is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named corral".

This tutorial is written for Corral 0.3 and Python 3.4 or later. If the Corral version doesn't match, you can refer to the tutorial for your version of Corral by using the version switcher at the bottom right corner of this page, or update Corral to the newest version. If you are still using Python 2.7, you will need to adjust the code samples slightly, as described in comments.

See *How to install Corral* for advice on how to remove older versions of Corral and install a newer one.

Where to get help:

If you're having trouble going through this tutorial, please post a message to https://github.com/toros-astro/corral to chat with other Corral users who might be able to help.

Creating a project

If this is your first time using Corral, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Corral *pipeline* – a collection of settings for an instance of Corral, including database configuration, Corral-specific options and pipeline-specific settings.

From the command line, `cd` into a directory where you'd like to store your code, then run the following command:

```
$ corral create my_pipeline
```

This will create a `my_pipeline` directory in your current directory.

Note: You'll need to avoid naming projects after built-in Python or Corral components. In particular, this means you should avoid using names like `corral` (which will conflict with Corral itself) or `test` (which conflicts with a built-in Python package). In most cases Corral must forbid the use of most commons names.

Let's look at what `create` created:

```
in_corral.py
my_pipeline/
├── __init__.py
├── settings.py
├── pipeline.py
├── models.py
├── load.py
├── steps.py
├── alerts.py
└── commands.py
```

These files are:

- `in_corral.py`: This is the acces point to your pipeline, and it allows commands to be executed inside the pipeline's environment.
- The inner `my_pipeline/` directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. `my_pipeline.models`).
- `my_pipeline/__init__.py`: An empty file that tells Python that this directory should be considered a Python package. (Read more about packages in the official Python docs if you're a Python beginner.)
- `my_pipeline/settings.py`: Settings/configuration for this Corral project.
- `my_pipeline/models.py`: This is the file that contains the entities (or tables) that are stored in the pipeline's database.
- `my_pipeline/pipeline.py`: This is the suggested file to globally configure the pipeline "on execution time".
- `my_pipeline/load.py`: This is where the pipeline's Loader lives. This would be the entry point for raw data to the pipeline stream, before going through any defined Steps.
- `my_pipeline/steps.py`: Every pipeline's step should be in this module, being this module one of the most important for data stream handling.
- `my_pipeline/alerts.py`: Inside this module the Alerts define the user custom communication channel to report expected results (a email for instance).
- `my_pipeline/commands.py`: Used to add custom console commands, specific for the pipeline.

4.2.2 Tutorial - Part #2 - Models

Study case: Iris Pipeline

We will carry out a simple exercise, using our recently initialized pipeline to develop a pipeline for statistic calculations of the famous Fisher Iris Dataset.

The plan is to obtain information for each class of the Iris species (Setosa, Virginica, and Versicolor) calculated separately, seizing the multi-processing of 3 cores at a time.

Finally we will set-up some alerts, just to let us know if any expected results are obtained.

We will define some commands as well, to check the pipeline general status.

Downloading the Data

First of all we need to download the csv file, with the raw data to feed the pipeline. We can get it from https://github.com/toros-astro/corral/raw/master/datasets/iris.csv and copy it inside the `my_pipeline` directory.

If we take a glance at our files at this point, it should look like:

```
in_corral.py
my_pipeline/
├── __init__.py
├── iris.csv
├── settings.py
├── pipeline.py
├── models.py
├── load.py
├── steps.py
├── alerts.py
└── commands.py
```

Basic Configuration

First thing to do is to edit `settings.py`.

A thing we need to be able to do, is finding paths dynamically, so we import the *os* module. The import should look like

```
import logging
import os
```

The `CONNECTION` variable specifies the *RFC-1738* format (used by SQLAlchemy) for database connection. Default should look something like this:

```
CONNECTION = "sqlite:///my_pipeline-dev.db"
```

With this instruction, a file `pipeline-dev.db` will be created in the same directory where `in_corral.py` is located, containing the SQLite database that we just defined.

See also:

For more information regarding other databases, you can search the SQLAlchemy documentation at: http://docs.sqlalchemy.org/en/latest/core/engines.html

At the end of the file we will add the following lines

```
PATH = os.path.abspath(os.path.dirname(__file__))
IRIS_PATH = os.path.join(PATH, "iris.csv")
```

First line stores in the variable `PATH` the directory where `settings.py` is located. The second line just creates a path to the file *iris.csv* that we downloaded before.

The Models

Now our pipeline needs to know the looks of our data stored in the database.

In `my_pipeline/models.py` file, we delete the `Example` class. Then we modify the file to look just like this:

```
class Name(db.Model):

    __tablename__ = 'Name'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True)


class Observation(db.Model):

    __tablename__ = 'Observation'

    id = db.Column(db.Integer, primary_key=True)

    name_id = db.Column(
        db.Integer, db.ForeignKey('Name.id'), nullable=False)
    name = db.relationship("Name", backref=db.backref("observations"))

    sepal_length = db.Column(db.Float, nullable=False)
    sepal_width = db.Column(db.Float, nullable=False)
    petal_length = db.Column(db.Float, nullable=False)
    petal_width = db.Column(db.Float, nullable=False)
```

As we can see, the `Name` and `Observation` classes inherit from `db.Model`, and by doing so, we let Corral know that these are tables in our database.

The `Name` model will be in charge of storing every different name on our dataset. Let's remember that the dataset has three different types of Iris flowers: *setosa*, *versicolor* and *virginica*, which will translate to three different instances of this model. In this same class we have only three attributes. The first one, `__tablename__`, will determine the name of the table that will be created on the database to make our data persistent (*Name* in our case). `id` is a column on the *Name* table for the primary key, with an integer type. Finally, the column `name` will hold the name of the species itself, with a maximum length of 50 characters, and this name cannot repeat across the column.

On the other hand, the model `Observation` has, besides the attributes `__tablename__` and `id`, references to the model `Name` (the attributes `name_id` and `name`). This implies that each instance of this table must have a name and 4 other columns with floating point numbers to hold the other 4 columns of the dataset.

Note: The models are models of the SQLAlchemy ORM in every sense; and `db.Model` is a declarative_base

To learn more about SQLAlchemy ORM please refer to their documentation on http://docs.sqlalchemy.org/en/rel_1_1/orm/tutorial.html

---

Note: When we execute the line `from corral import db`, we have available inside the `db` namespace, the namespaces for `sqlalchemy`, `sqlalchemy.orm` and `sqlalchemy_utils`.

Learn more about sqlalchemy_utils on: http://sqlalchemy-utils.readthedocs.org

---

To create the database, we need to execute the command:

```
$ python in_corral.py createdb
```

After a confirmation question, the output should look like this:

```
Do you want to create the database [Yes/no]? yes
[my_pipeline-INFO @ 2016-01-08 01:44:01,027] SELECT CAST('test plain returns' AS␣
↪VARCHAR(60)) AS anon_1
```

(continues on next page)

(continued from previous page)

```
[my_pipeline-INFO @ 2016-01-08 01:44:01,028] ()
[my_pipeline-INFO @ 2016-01-08 01:44:01,029] SELECT CAST('test unicode returns' AS␣
↪VARCHAR(60)) AS anon_1
[my_pipeline-INFO @ 2016-01-08 01:44:01,029] ()
[my_pipeline-INFO @ 2016-01-08 01:44:01,031] PRAGMA table_info("Observation")
[my_pipeline-INFO @ 2016-01-08 01:44:01,031] ()
[my_pipeline-INFO @ 2016-01-08 01:44:01,060] PRAGMA table_info("Name")
[my_pipeline-INFO @ 2016-01-08 01:44:01,060] ()
[my_pipeline-INFO @ 2016-01-08 01:44:01,061]
CREATE TABLE "Name" (
    id INTEGER NOT NULL,
    name VARCHAR(50),
    PRIMARY KEY (id),
    UNIQUE (name)
)

[my_pipeline-INFO @ 2016-01-08 01:44:01,201] ()
[my_pipeline-INFO @ 2016-01-08 01:44:01,333] COMMIT
[my_pipeline-INFO @ 2016-01-08 01:44:01,334]
CREATE TABLE "Observation" (
    id INTEGER NOT NULL,
    name_id INTEGER NOT NULL,
    sepal_length FLOAT NOT NULL,
    sepal_width FLOAT NOT NULL,
    petal_length FLOAT NOT NULL,
    petal_width FLOAT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(name_id) REFERENCES "Name" (id)
)

[my_pipeline-INFO @ 2016-01-08 01:44:01,334] ()
[my_pipeline-INFO @ 2016-01-08 01:44:01,467] COMMIT
```

We can read in the output, the SQL instructions used to create the tables to make our models persistent, plus some extra tables used as support by corral, like `__corral_alerted__`

We can explore our recently created empty database, with the command `python in_corral.py dbshell`

```
$ python in_corral.py dbshell
Connected to: Engine(sqlite:///my_pipeline-dev.db)
Type 'exit;' or '<CTRL> + <D>' for exit the shell

SQL> select * from sqlite_master where type = 'table' and name != '__corral_alerted__
↪';
+-------+------------+------------+----------+-------------------------------------
↪---------------+
| type  |    name    |  tbl_name  | rootpage |                         sql         ␣
↪               |
+=======+============+============+==========+=============================================
| table | Name       | Name       | 2        | CREATE TABLE "Name" (               ␣
↪               |
|       |            |            |          |         id INTEGER NOT NULL,        ␣
↪               |
|       |            |            |          |         name VARCHAR(50),           ␣
↪               |
|       |            |            |          |         PRIMARY KEY (id),           ␣
↪               |
```

(continues on next page)

(continued from previous page)

```
|       |             |             |          |         UNIQUE (name)                                  ␣
↪               |
|       |             |             |          | )                                                      ␣
↪               |
| table | Observation | Observation | 5        | CREATE TABLE "Observation" (                            ␣
↪               |
|       |             |             |          |         id INTEGER NOT NULL,                           ␣
↪               |
|       |             |             |          |         name_id INTEGER NOT NULL,                      ␣
↪               |
|       |             |             |          |         sepal_length FLOAT NOT NULL,                   ␣
↪               |
|       |             |             |          |         sepal_width FLOAT NOT NULL,                    ␣
↪               |
|       |             |             |          |         petal_length FLOAT NOT NULL,                   ␣
↪               |
|       |             |             |          |         petal_width FLOAT NOT NULL,                    ␣
↪               |
|       |             |             |          |         PRIMARY KEY (id),                              ␣
↪               |
|       |             |             |          |         FOREIGN KEY(name_id)␣
↪REFERENCES "Name" (id) |
|       |             |             |          | )                                                      ␣
↪               |
+-------+-------------+-------------+----------+-----------------------------------------
↪---------------+
SQL>
```

4.2.3 Tutorial - Part #3 - Loaders

Loading Data on the Stream: Loader

At this point we already have:

- Data in a file `iris.csv`.
- The `settings.py` containing the path to the file.
- Models already defined (in `models.py`) to store *Name* and the *Observations*

Now the next step is to parse data in the `iris.csv` on the *modelos* working with Corral's **Loader**.

The loaders idea is to work as an entry point for raw data to the pipeline processing chain. Opposed to the *Steps* (on the next tutorial section), the *Loaders* are not restricted by the defined models of our stream.

As everythin in Corral, the **Loaders** are defined as a Class, suggested to be in a separated file named `load.py` of your project. Also this Class must be registered in the `settings.py` file.

Reading iris.csv data

Python can work with CSV files module https://docs.python.org/3.5/library/csv.html which contains a parser capable to transform each row in the file into a dictionary with it's keys as column names

So for instance

```
$ python in_corral.py shell # open a shell inside the pipeline environment
LOAD: Name, Observation (my_pipeline.models)
LOAD: session (sqlalchemy.orm.session)
------------------------------------------------------------------------------

# import the settings to load the IRIS_PATH
>>> from corral.conf import settings
>>> settings.IRIS_PATH
'path/to/my_pipeline/iris.csv'

# import the csv handler module and also read the file with it and print
# the output into the console
>>> import csv
>>> for row in csv.DictReader(open(settings.IRIS_PATH)):
...     print(row)
...
{'SepalLength': '5.1', 'PetalLength': '1.4', 'PetalWidth': '0.2', 'SepalWidth': '3.5',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.9', 'PetalLength': '1.4', 'PetalWidth': '0.2', 'SepalWidth': '3.0',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.7', 'PetalLength': '1.3', 'PetalWidth': '0.2', 'SepalWidth': '3.2',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.6', 'PetalLength': '1.5', 'PetalWidth': '0.2', 'SepalWidth': '3.1',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '5.0', 'PetalLength': '1.4', 'PetalWidth': '0.2', 'SepalWidth': '3.6',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '5.4', 'PetalLength': '1.7', 'PetalWidth': '0.4', 'SepalWidth': '3.9',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.6', 'PetalLength': '1.4', 'PetalWidth': '0.3', 'SepalWidth': '3.4',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '5.0', 'PetalLength': '1.5', 'PetalWidth': '0.2', 'SepalWidth': '3.4',
↪ 'Name': 'Iris-setosa'}
# ... MANY MORE LINES ... #
```

To write the loader what we should do is to open the file `pipeline/load.py` which should look like this:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# ======================================================================
# DOCS
# ======================================================================

"""pipeline main loader

"""


# ======================================================================
# IMPORTS
# ======================================================================

from corral import run


# ======================================================================
# LOADER
```

(continues on next page)

(continued from previous page)

```
# ============================================================================

class Loader(run.Loader):

    def generate(self):
        # write your logic here
        pass
```

First we need to import the python module `csv`, the `settings` from corral and import from our pipeline the models module, in order to generate them using the loader. With all this the import block should have this looks:

```
# ============================================================================
# IMPORTS
# ============================================================================

import csv

from corral import run
from corral.conf import settings

from my_pipeline import models
```

The `Loader.generate()` method now could start reading the csv file and screen print it, as like we did in the interactive session:

```
class Loader(run.Loader):

    def generate(self):
        for row in csv.DictReader(open(settings.IRIS_PATH)):
            print(row)
```

Now if we go to the command line and execute

```
$ python in_corral.py load
```

We will get an output just like the following:

```
[my_pipeline-INFO @ 2016-01-10 17:59:00,393] Executing loader '<class 'my_pipeline.
↪load.Loader'>' #1
{'SepalLength': '5.1', 'PetalLength': '1.4', 'PetalWidth': '0.2', 'SepalWidth': '3.5',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.9', 'PetalLength': '1.4', 'PetalWidth': '0.2', 'SepalWidth': '3.0',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.7', 'PetalLength': '1.3', 'PetalWidth': '0.2', 'SepalWidth': '3.2',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.6', 'PetalLength': '1.5', 'PetalWidth': '0.2', 'SepalWidth': '3.1',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '5.0', 'PetalLength': '1.4', 'PetalWidth': '0.2', 'SepalWidth': '3.6',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '5.4', 'PetalLength': '1.7', 'PetalWidth': '0.4', 'SepalWidth': '3.9',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.6', 'PetalLength': '1.4', 'PetalWidth': '0.3', 'SepalWidth': '3.4',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '5.0', 'PetalLength': '1.5', 'PetalWidth': '0.2', 'SepalWidth': '3.4',
↪ 'Name': 'Iris-setosa'}
{'SepalLength': '4.4', 'PetalLength': '1.4', 'PetalWidth': '0.2', 'SepalWidth': '2.9',
↪ 'Name': 'Iris-setosa'}
```

(continues on next page)

(continued from previous page)

```
{'SepalLength': '4.9', 'PetalLength': '1.5', 'PetalWidth': '0.1', 'SepalWidth': '3.1',
↪ 'Name': 'Iris-setosa'}
# ... MANY MORE LINES ... #
{'SepalLength': '6.2', 'PetalLength': '5.4', 'PetalWidth': '2.3', 'SepalWidth': '3.4',
↪ 'Name': 'Iris-virginica'}
{'SepalLength': '5.9', 'PetalLength': '5.1', 'PetalWidth': '1.8', 'SepalWidth': '3.0',
↪ 'Name': 'Iris-virginica'}
[my_pipeline-INFO @ 2016-01-10 17:59:00,396] Done Loader '<class 'my_pipeline.load.
↪Loader'>' #1
```

Which tells us that the loader is able to acces the `iris.csv` file, and printing its content.

As a matter of order and safety it is convenient that files close explicitly just one time per process. To get this we could just redefine the `Loader` method's `setup` and `teardown`.

`setup` is executed just before `generate` and it is the best place to open our file. On the other hand `teardown` gets information related to the error state of the `generate` method, and runs just after this one ends. The simplest way to implement this is the following:

```
class Loader(run.Loader):

    def setup(self):
        # we open the file and assign it to an instance variable
        self.fp = open(settings.IRIS_PATH)

    def teardown(self, *args):
        # checking that the file is really open
        if self.fp and not self.fp.closed:
            self.fp.close()

    def generate(self):
        # now we make use of "self.fp" for the reader
        for row in csv.DictReader(self.fp):
            print(row)
```

For the sake of simplicity now we split the processing into two sides:

1. A method named `get_name_instance` which receives the row as a parameter and returns a `my_pipeline.models.Name` instance referred to the *name* of such file (*Iris-virginica*, *Iris-versicolor*, or *Iris-setosa*). Something to take into account is that every time a name is non existant this method must create a new one and to store this model before returning it.
2. A method named `store_observation` which receives the row as a parameter, and also the instance of `my_pipeline.models.Name` just created by the previous model. This method just needs to return the instance and deliver it to the loader without saving it.

Warning: This tutorial is going to assume a certain level of knowledge in sessions, queries from SQLAlchemy. If any doubts arise, please go to orm tutorial

First of all we define the method `get_name_instance`

```
def get_name_instance(self, row):
    name = self.session.query(models.Name).filter(
        models.Name.name == row["Name"]).first()
```

(continues on next page)

(continued from previous page)

```
    # if exists we don't need to create one
    if name is None:
        name = models.Name(name=row["Name"])

        # we need to add the new instance and save it
        self.save(name)
        self.session.commit()

    return name
```

now `store_observation`:

```
def store_observation(self, row, name):
    return models.Observation(
        name=name,
        sepal_length=row["SepalLength"], sepal_width=row["SepalWidth"],
        petal_length=row["PetalLength"], petal_width=row["PetalWidth"])
```

Finally the `generate` method would be defined as:

```
def generate(self):
    # now we use the "self.fp" for the reader
    for row in csv.DictReader(self.fp):
        name = self.get_name_instance(row)
        obs = self.store_observation(row, name)
        yield obs
```

In the very last line with the `yield` command, we deliver the instance created by `store_observation` to corral so it would be persisted when the time comes.

Warning: Bare in mind that `generate` *by default* can only return `None` or a *models* instance *iterator* or a single *model*. If you wish for it to generate another object it is necessary to redefine the `validate` method which is not treated on this tutorial.

Finally the loader should be defined as:

```
class Loader(run.Loader):

    def setup(self):
        # we open the file and assign it to an instance variable
        self.fp = open(settings.IRIS_PATH)

    def teardown(self, *args):
        # checking that the file is really open
        if self.fp and not self.fp.closed:
            self.fp.close()

    def get_name_instance(self, row):
        name = self.session.query(models.Name).filter(
            models.Name.name == row["Name"]).first()

        # if exists we need don't need to create one
        if name is None:
            name = models.Name(name=row["Name"])
```

(continues on next page)

(continued from previous page)

```
            # we need to add the new instance and save it
            self.save(name)
            self.session.commit()

        return name

    def store_observation(self, row, name):
        return models.Observation(
            name=name,
            sepal_length=row["SepalLength"], sepal_width=row["SepalWidth"],
            petal_length=row["PetalLength"], petal_width=row["PetalWidth"])

    def generate(self):
        # now we make use of "self.fp" for the reader
        for row in csv.DictReader(self.fp):
            name = self.get_name_instance(row)
            obs = self.store_observation(row, name)
            yield obs
```

Note: If you wish to register another name for the loader class, just update the value of the `LOADER` variable in `settings.py`.

Now when we run

```
$ python in_corral load
```

the result will be a list of sql commands that should look like this:

```
...
[my_pipeline-INFO @ 2016-01-10 19:10:21,800] ('Iris-setosa', 1, 0)
[my_pipeline-INFO @ 2016-01-10 19:10:21,801] INSERT INTO "Observation" (name_id,␣
↪sepal_length, sepal_width, petal_length, petal_width) VALUES (?, ?, ?, ?, ?)
[my_pipeline-INFO @ 2016-01-10 19:10:21,801] (1, 4.6, 3.4, 1.4, 0.3)
[my_pipeline-INFO @ 2016-01-10 19:10:21,802] SELECT "Name".id AS "Name_id", "Name".
↪name AS "Name_name"
FROM "Name"
WHERE "Name".name = ?
 LIMIT ? OFFSET ?
[my_pipeline-INFO @ 2016-01-10 19:10:21,802] ('Iris-setosa', 1, 0)
[my_pipeline-INFO @ 2016-01-10 19:10:21,804] INSERT INTO "Observation" (name_id,␣
↪sepal_length, sepal_width, petal_length, petal_width) VALUES (?, ?, ?, ?, ?)
[my_pipeline-INFO @ 2016-01-10 19:10:21,804] (1, 5.0, 3.4, 1.5, 0.2)
...
```

We can explore the loaded data with:

```
$ python in_corral.py dbshell
Connected to: Engine(sqlite:///my_pipeline-dev.db)
Type 'exit;' or '<CTRL> + <D>' for exit the shell

SQL> select * from observation limit 10;
+----+---------+--------------+-------------+--------------+-------------+
| id | name_id | sepal_length | sepal_width | petal_length | petal_width |
+====+=========+==============+=============+==============+=============+
| 1  | 1       | 5.100        | 3.500       | 1.400        | 0.200       |
```

(continues on next page)

(continued from previous page)

```
| 2  | 1       | 4.900        | 3           | 1.400        | 0.200       |
| 3  | 1       | 4.700        | 3.200       | 1.300        | 0.200       |
| 4  | 1       | 4.600        | 3.100       | 1.500        | 0.200       |
| 5  | 1       | 5            | 3.600       | 1.400        | 0.200       |
| 6  | 1       | 5.400        | 3.900       | 1.700        | 0.400       |
| 7  | 1       | 4.600        | 3.400       | 1.400        | 0.300       |
| 8  | 1       | 5            | 3.400       | 1.500        | 0.200       |
| 9  | 1       | 4.400        | 2.900       | 1.400        | 0.200       |
| 10 | 1       | 4.900        | 3.100       | 1.500        | 0.100       |
+----+---------+--------------+-------------+--------------+-------------+
SQL>
```

Or more easily with Python:

```
>>> for obs in session.query(Observation).all():
...     print(obs)
...
[my_pipeline-INFO @ 2016-01-10 19:24:20,555] SELECT CAST('test plain returns' AS␣
↪VARCHAR(60)) AS anon_1
[my_pipeline-INFO @ 2016-01-10 19:24:20,556] ()
[my_pipeline-INFO @ 2016-01-10 19:24:20,556] SELECT CAST('test unicode returns' AS␣
↪VARCHAR(60)) AS anon_1
[my_pipeline-INFO @ 2016-01-10 19:24:20,556] ()
[my_pipeline-INFO @ 2016-01-10 19:24:20,557] BEGIN (implicit)
[my_pipeline-INFO @ 2016-01-10 19:24:20,558] SELECT "Observation".id AS "Observation_
↪id", "Observation".name_id AS "Observation_name_id", "Observation".sepal_length AS
↪"Observation_sepal_length", "Observation".sepal_width AS "Observation_sepal_width",
↪"Observation".petal_length AS "Observation_petal_length", "Observation".petal_width␣
↪AS "Observation_petal_width"
FROM "Observation"
[my_pipeline-INFO @ 2016-01-10 19:24:20,558] ()
<my_pipeline.models.Observation object at 0x7fd14f45ee90>
<my_pipeline.models.Observation object at 0x7fd14f45e9d0>
<my_pipeline.models.Observation object at 0x7fd14f45eb50>
<my_pipeline.models.Observation object at 0x7fd14f45e950>

>>> for name in session.query(Name).all():
...     print(name)
...
[my_pipeline-INFO @ 2016-01-10 19:26:01,907] SELECT "Name".id AS "Name_id", "Name".
↪name AS "Name_name"
FROM "Name"
[my_pipeline-INFO @ 2016-01-10 19:26:01,907] ()
<my_pipeline.models.Name object at 0x7fd14f414a50>
<my_pipeline.models.Name object at 0x7fd14f414b10>
<my_pipeline.models.Name object at 0x7fd14f414bd0>
```

This output could be improved, since it doesn't give much information. To do this, we can redefine the `__repr__` method for each model (https://docs.python.org/2/reference/datamodel.html#object.__repr__)

Improving the interactive session instance feedback

We can define the `__repr__` of `Name` as:

```
class Name(db.Model):

    ...

    def __repr__(self):
        return "<Name '{}' {}>".format(self.name, self.id)
```

and of `Observation` like this:

```
class Observation(db.Model):

    ...

    def __repr__(self):
        return "<Observation ({}, {}, {}, {}, {}) {}>".format(
            self.name.name,
            self.sepal_length, self.sepal_width,
            self.petal_length, self.petal_width, self.id)
```

```
$ python in_corral.py shell --shell plain
LOAD: Name, Observation (my_pipeline.models)
LOAD: session (sqlalchemy.orm.session)
------------------------------------------------------------------------------
>>> for obs in session.query(Observation).all():
...     print(obs)
...
<Observation (Iris-setosa, 5.1, 3.5, 1.4, 0.2) 1>
<Observation (Iris-setosa, 4.9, 3.0, 1.4, 0.2) 2>
<Observation (Iris-setosa, 4.7, 3.2, 1.3, 0.2) 3>

# Or we could search for every versicolor
>>> name_versicolor = session.query(Name).filter(Name.name=="Iris-versicolor").first()
>>>  name_versicolor.observations
...
[<Observation (Iris-versicolor, 7.0, 3.2, 4.7, 1.4) 51>,
 <Observation (Iris-versicolor, 6.4, 3.2, 4.5, 1.5) 52>,
 <Observation (Iris-versicolor, 6.9, 3.1, 4.9, 1.5) 53>,
 <Observation (Iris-versicolor, 5.5, 2.3, 4.0, 1.3) 54>,
 <Observation (Iris-versicolor, 6.5, 2.8, 4.6, 1.5) 55>,
 ...]
```

4.2.4 Tutorial - Part #4 - Steps

Steps: Processing Data

After we execute the line `python in_corral load` we have the iris data loaded in our database and now we want to calculate the mean, minimum and maximum values for `sepal_length`, `sepal_width`, `petal_length` and `petal_width` in parallel for each species.

Warning: All throughout this tutorial we have used SQLite as our database. SQLite does not support concurrency. Keep in mind this is just an excercise and a real pipeline should use a database like PostgreSQL, MySQL, Oracle or something even more powerful like Hive

A Model for the Statistics

To hold the statistics, we will define a model with the three statistical measures for the four observed properties of the species. It will also hold a reference to the Iris species to which it belong (a relation to the `Name` table.)

To do so, we add at the end of `my_pipeline/models.py`, the class

```
class Statistics(db.Model):

    __tablename__ = 'Statistics'

    id = db.Column(db.Integer, primary_key=True)

    name_id = db.Column(
        db.Integer, db.ForeignKey('Name.id'), nullable=False, unique=True)
    name = db.relationship(
        "Name", backref=db.backref("statistics"), uselist=False)

    mean_sepal_length = db.Column(db.Float, nullable=False)
    mean_sepal_width = db.Column(db.Float, nullable=False)
    mean_petal_length = db.Column(db.Float, nullable=False)
    mean_petal_width = db.Column(db.Float, nullable=False)

    min_sepal_length = db.Column(db.Float, nullable=False)
    min_sepal_width = db.Column(db.Float, nullable=False)
    min_petal_length = db.Column(db.Float, nullable=False)
    min_petal_width = db.Column(db.Float, nullable=False)

    max_sepal_length = db.Column(db.Float, nullable=False)
    max_sepal_width = db.Column(db.Float, nullable=False)
    max_petal_length = db.Column(db.Float, nullable=False)
    max_petal_width = db.Column(db.Float, nullable=False)

    def __repr__(self):
        return "<Statistics of '{}'>".format(self.name.name)
```

If you have already read our last tutorial, the only differences this model has with the previous ones are the parameters `unique=True` and `userlist=False` on the lines where we define the relation. These are used to enforce that each instance of `Name` has one and only one instance of `Statistics`.

To create the table we execute once again on the command line `python in_corral createdb` and only the new table will be crated without changing the shape and form of the previous ones.

The Steps

We will create four steps in the `my_pypeline/steps.py` module.

#. Step 1: Creating Statistics for each Name

First, uncomment on the import section the line `# from . import models`; and then edit the class `MyStep` so that it looks like the following:

```
class StatisticsCreator(run.Step):

    model = models.Name
```

(continues on next page)

(continued from previous page)

```
    conditions = []

    def process(self, name):
        stats = self.session.query(models.Statistics).filter(
            models.Statistics.name_id==name.id).first()
        if stats is None:
            yield models.Statistics(
                name_id=name.id,
                mean_sepal_length=0., mean_sepal_width=0.,
                mean_petal_length=0., mean_petal_width=0.,
                min_sepal_length=0., min_sepal_width=0.,
                min_petal_length=0., min_petal_width=0.,
                max_sepal_length=0., max_sepal_width=0.,
                max_petal_length=0., max_petal_width=0.)
```

This step's goal is to create an instance of `Statistics` for each different name it finds on the `Name` table.

Notice that we let the *Step* know in the variable `model` that it will be working with unconditioned instances of the model `Name`. Corral will sequentially send the stored (by the Loader) instances, that meet the conditions (all of the instances in our case).

The `process()` method receives each instance of `Name` and if there is no associated instance of `Statistic`, it will create one with all the values set to *0*, yielding back the control to corral (with `yield`).

#. Step 2: Calculating Statistics for "Iris-Setosa"

If we create a Step `SetosaStatistics` and we assign to its model variable the class `Statistics` and we add the `conditions`:

```
conditions = [
        models.Statistics.name.has(name="Iris-setosa"),
        models.Statistics.mean_sepal_length==0.]
```

we will create a step that only calculates the statistics of **Iris-setosa** if they were not previously calculated (the mean for `sepal_length` is `0.`)

The `process()` method will be passed by parameter said instance of `Statistics`. To fill the statistics out, the complete code for this step will be:

```
class SetosaStatistics(run.Step):

    model = models.Statistics
    conditions = [
        models.Statistics.name.has(name="Iris-setosa"),
        models.Statistics.mean_sepal_length==0.]

    def process(self, stats):
        sepal_length, sepal_width, petal_length, petal_width = [], [], [], []
        for obs in stats.name.observations:
            sepal_length.append(obs.sepal_length)
            sepal_width.append(obs.sepal_width)
            petal_length.append(obs.petal_length)
            petal_width.append(obs.petal_width)

        stats.mean_sepal_length = sum(sepal_length) / len(sepal_length)
        stats.mean_sepal_width = sum(sepal_width) / len(sepal_width)
```

(continues on next page)

(continued from previous page)

```
        stats.mean_petal_length = sum(petal_length) / len(petal_length)
        stats.mean_petal_width = sum(petal_width) / len(petal_width)

        stats.min_sepal_length = min(sepal_length)
        stats.min_sepal_width = min(sepal_width)
        stats.min_petal_length = min(petal_length)
        stats.min_petal_width = min(petal_width)

        stats.max_sepal_length = max(sepal_length)
        stats.max_sepal_width = max(sepal_width)
        stats.max_petal_length = max(petal_length)
        stats.max_petal_width = max(petal_width)
```

#. Step 3 and 4: Calculating Statistics for "Iris-Virginica" and "Iris-Versicolor"

The last two steps are exactly the same as the previous ones, except for the variables `model` and `conditions`.

```
class VersicolorStatistics(run.Step):

    model = models.Statistics
    conditions = [
        models.Statistics.name.has(name="Iris-versicolor"),
        models.Statistics.mean_sepal_length==0.]

    def process(self, stats):
        # SAME CODE AS SetosaStatistics.process


class VirginicaStatistics(run.Step):

    model = models.Statistics
    conditions = [
        models.Statistics.name.has(name="Iris-virginica"),
        models.Statistics.mean_sepal_length==0.]

    def process(self, stats):
        # SAME CODE AS SetosaStatistics.process
```

#. Step 5: Add the new steps to `settings.STEPS`

The last piece is to make your pipeline aware of the new steps. For this, you need to add the full python path to the `STEPS` list inside the settings.py file.

```
# Pipeline processor steps
STEPS = [
    "my_pipeline.steps.StatisticsCreator",
    "my_pipeline.steps.SetosaStatistics",
    "my_pipeline.steps.VirginicaStatistics",
    "my_pipeline.steps.VersicolorStatistics"]
```

Finally you can inspect the registered steps with the `lssteps` command

```
$python in_corral.py lssteps
+----------------------+---------+---------+
|      Step Class      | Process | Groups  |
+======================+=========+=========+
| SetosaStatistics     | 1       | default |
| StatisticsCreator    | 1       | default |
| VersicolorStatistics | 1       | default |
| VirginicaStatistics  | 1       | default |
+----------------------+---------+---------+
  TOTAL PROCESSES: 4
  DEBUG PROCESS: Enabled
```

Also note that (by default) every step is on the **default** group.

Note: The command `python in_corral groups` shows all available groups in steps and alerts.

Running The Steps

The main command to run the corral steps is **run**.

when you execute `python in_corral run` all the steps are executed asynchronous. If for some particular case you need to run the steps sequentially (in the same order of `settings.STEPS`) you can add the `--sync` flag.

Warning: By design, SQLite is not capable to serve as a multiprocess database, so it is highly recommended to run the steps with the `--sync` flag.

Here is a **run** example output

```
$ python in_corral.py run --sync
[INFO] Executing step '<class 'my_pipeline.steps.SetosaStatistics'>' #1
[INFO] SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
[INFO] ()
[INFO] SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
[INFO] ()
[INFO] BEGIN (implicit)
[INFO] SELECT "Statistics".id AS "Statistics_id", "Statistics".name_id AS "Statistics_
↪name_id", "Statistics".mean_sepal_length AS "Statistics_mean_sepal_length",
↪"Statistics".mean_sepal_width AS "Statistics_mean_sepal_width", "Statistics".mean_
↪petal_length AS "Statistics_mean_petal_length", "Statistics".mean_petal_width AS
↪"Statistics_mean_petal_width", "Statistics".min_sepal_length AS "Statistics_min_
↪sepal_length", "Statistics".min_sepal_width AS "Statistics_min_sepal_width",
↪"Statistics".min_petal_length AS "Statistics_min_petal_length", "Statistics".min_
↪petal_width AS "Statistics_min_petal_width", "Statistics".max_sepal_length AS
↪"Statistics_max_sepal_length", "Statistics".max_sepal_width AS "Statistics_max_
↪sepal_width", "Statistics".max_petal_length AS "Statistics_max_petal_length",
↪"Statistics".max_petal_width AS "Statistics_max_petal_width"
FROM "Statistics"
WHERE (EXISTS (SELECT 1
FROM "Name"
WHERE "Name".id = "Statistics".name_id AND "Name".name = ?)) AND "Statistics".mean_
↪sepal_length = ?
[INFO] ('Iris-setosa', 0.0)
[INFO] COMMIT
```

(continues on next page)

(continued from previous page)

```
[INFO] Done Step '<class 'pipeline.steps.SetosaStatistics'>' #1
[INFO] Executing step '<class 'pipeline.steps.StatisticsCreator'>' #1
[INFO] BEGIN (implicit)
[INFO] SELECT "Name".id AS "Name_id", "Name".name AS "Name_name"
FROM "Name"
...
```

Selective Steps Runs By Name and Groups

In some cases it is useful to run only a single or a group of steps.

Run by Name

You can run a single step by using the `--steps|-s` flag followed by the class-names of the steps you want to run.

```
$ python in_corral.py run --steps SetosaStatistics VersicolorStatistics
[INFO] Executing step '<class 'irispl.steps.SetosaStatistics'>' #1
[INFO] Executing step '<class 'irispl.steps.VersicolorStatistics'>' #1
...
```

Run by Groups

One of the most important concepts with Corral steps is the notion of groups.

Certain steps can be grouped together by adding a `groups` attribute to a Step class. For example, if we want to add the tree statistics calculators steps to a `statistics` group, we'd write:

```
class SetosaStatistics(run.Step):
    model = models.Statistics
    conditions = [
        models.Statistics.name.has(name="Iris-versicolor"),
        models.Statistics.mean_sepal_length==0.]
    groups = ["default", "statistics"]

    ...


class VersicolorStatistics(run.Step):

    model = models.Statistics
    conditions = [
        models.Statistics.name.has(name="Iris-versicolor"),
        models.Statistics.mean_sepal_length==0.]
    groups = ["default", "statistics"]

    ...


class VirginicaStatistics(run.Step):

    model = models.Statistics
    conditions = [
```

(continues on next page)

(continued from previous page)

```
        models.Statistics.name.has(name="Iris-virginica"),
        models.Statistics.mean_sepal_length==0.]
    groups = ["default", "statistics"]
```

You can check the changes on the column `Groups` by running `lssteps` again

```
$ python in_corral.py lssteps
+----------------------+---------+--------------------+
|      Step Class      | Process |       Groups       |
+======================+=========+====================+
| SetosaStatistics     | 1       | default:statistics |
| StatisticsCreator    | 1       | default            |
| VersicolorStatistics | 1       | default:statistics |
| VirginicaStatistics  | 1       | default:statistics |
+----------------------+---------+--------------------+
  TOTAL PROCESSES: 4
  DEBUG PROCESS: Enabled
```

You can also list only the steps of a particular group with the `--groups|-g` flag

```
$ python in_corral.py lssteps -g statistics
+----------------------+---------+--------------------+
|      Step Class      | Process |       Groups       |
+======================+=========+====================+
| SetosaStatistics     | 1       | default:statistics |
| VersicolorStatistics | 1       | default:statistics |
| VirginicaStatistics  | 1       | default:statistics |
+----------------------+---------+--------------------+
  TOTAL PROCESSES: 3
  DEBUG PROCESS: Enabled
```

Finally, you can run the group of your choice with the `--step-groups|--sg` flag on the **run** command

```
$ python in_corral.py run -sg statistics
[INFO] Executing step '<class 'irispl.steps.SetosaStatistics'>' #1
[INFO] Executing step '<class 'irispl.steps.VersicolorStatistics'>' #1
[INFO] Executing step '<class 'irispl.steps.VirginicaStatistics'>' #1
...
```

As you can see, the `StatisticsCreator` step didn't run.

4.2.5 Tutorial - Part #5 - Alerts

Alerts: Inform about some desired State

In a single phrase:

```
An Alert is a step that does not store information, but it will send it to
some other place, away from the pipeline.
```

In our infrastructure, an Alert is a View in the MVC pattern, since it is responsible to inform some potential final user about some anomalous state (desired or not) within the pipeline data.

The idea behind alerts is to design them as steps, but to add them one or several destinations (Endpoints) on top; in the chosen models (?)

La idea detras de las alerts es diseñarlas como steps, pero ademas agregarles uno varios destinos (Endpoint); en los modelos escogidos por se serializen

Corral offers two default endpoints:

- `Email`: The model data is sent by email.
- `File`: The model data are written to a local file.

Creating an Alert

In our example, we will write an Alert that writes each statistics of the data to a file.

To do so, we edit the class MyAlert in `my_pipeline/alerts.py`

```
from corral import run
from corral.run import endpoints as ep

from . import models

class StatisticsAlert(run.Alert):

    model = models.Statistics
    conditions = []
    alert_to = [ep.File("statistics.log")]
```

An Alert's endpoints are added to the variable `alert_to`. The endpoint `File` only receives as a required parameter the path to the file to write to, and optional parameters `mode` and `enconding`. The mode parameter refers to the mode the file is opened (`a` append by default); `encoding` refers to the encoding of the file to open (`utf-8` by default).

Finally, the last step is editing the variable `ALERTS` in `settings.py` so that it contains our new alert.

```
# The alerts
ALERTS = ["irispl.alerts.StatisticsAlert"]
```

Once it's done, we can verify if out Alert is addded correctly by running the command `lsalerts`

```
$ python in_corral.py lsalerts
+-----------------+---------+---------+
|   Alert Class   | Process | Groups  |
+=================+=========+=========+
| StatisticsAlert | 1       | default |
+-----------------+---------+---------+
  TOTAL PROCESSES: 1
  DEBUG PROCESS: Enabled
```

To run the alert we just need to execute

```
$ python in_corral check-alerts
[INFO] Executing alert '<class 'irispl.alerts.StatisticsAlert'>' #1
[INFO] SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1
[INFO] ()
[INFO] SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1
[INFO] ()
[INFO] BEGIN (implicit)
[INFO] SELECT count(*) AS count_1
FROM (SELECT __corral_alerted__.model_ids AS __corral_alerted___model_ids
```

(continues on next page)

(continued from previous page)

```
FROM __corral_alerted__
WHERE __corral_alerted__.model_table = ? AND __corral_alerted__.alert_path = ?) AS␣
↪anon_1
[INFO] ('Statistics', 'irispl.alerts.StatisticsAlert')
[INFO] SELECT __corral_alerted__.model_ids AS __corral_alerted___model_ids
FROM __corral_alerted__
WHERE __corral_alerted__.model_table = ? AND __corral_alerted__.alert_path = ?
[INFO] ('Statistics', 'irispl.alerts.StatisticsAlert')
[INFO] SELECT "Statistics".id AS "Statistics_id", "Statistics".name_id AS "Statistics_
↪name_id", "Statistics".mean_sepal_length AS "Statistics_mean_sepal_length",
↪"Statistics".mean_sepal_width AS "Statistics_mean_sepal_width", "Statistics".mean_
↪petal_length AS "Statistics_mean_petal_length", "Statistics".mean_petal_width AS
↪"Statistics_mean_petal_width", "Statistics".min_sepal_length AS "Statistics_min_
↪sepal_length", "Statistics".min_sepal_width AS "Statistics_min_sepal_width",
↪"Statistics".min_petal_length AS "Statistics_min_petal_length", "Statistics".min_
↪petal_width AS "Statistics_min_petal_width", "Statistics".max_sepal_length AS
↪"Statistics_max_sepal_length", "Statistics".max_sepal_width AS "Statistics_max_
↪sepal_width", "Statistics".max_petal_length AS "Statistics_max_petal_length",
↪"Statistics".max_petal_width AS "Statistics_max_petal_width"
FROM "Statistics"
WHERE "Statistics".id NOT IN (?, ?, ?)
[INFO] (1, 2, 3)
[INFO] COMMIT
[INFO] Done Alert '<class 'irispl.alerts.StatisticsAlert'>' #1
```

If we now check the content of the *statistics.log* file, we'll see the following

```
$ cat statistics.log
[irispl-ALERT @ 2017-03-30T02:43:36.123542-15s] Check the object '<Statistics of
↪'Iris-setosa'>'
[irispl-ALERT @ 2017-03-30T02:43:36.124799-15s] Check the object '<Statistics of
↪'Iris-versicolor'>'
[irispl-ALERT @ 2017-03-30T02:43:36.126659-15s] Check the object '<Statistics of
↪'Iris-virginica'>'
```

As expected, we created a register of each created statistic. If we run the Alert again, we'll see that no more registers are added, since Corral keeps an internal record of the alerted models.

If we want to improve the alert message we can do so, redefining the method `render_alert()` of our Alert. This method receives three parameters:

- `utcnow` current date and time in UTC format.
- `endpoint` the endpoint to which we render the message.
- `obj` the object we alert about.

For instance, if we wanted to improve the message so that it informs us about all the statistics, we could write:

```
class StatisticsAlert(run.Alert):

    model = models.Statistics
    conditions = []
    alert_to = [ep.File("statistics.log")]

    def render_alert(self, utcnow, endpoint, obj):
        return """
            ALERT@{now}: {name}
```

(continues on next page)

(continued from previous page)

```
                - mean_sepal_length = {mean_sepal_length}
                - mean_sepal_width  = {mean_sepal_width}
                - mean_petal_length = {mean_petal_length}
                - mean_petal_width  = {mean_petal_width}
            ------------------------------------------------------
        """.rstrip().format(
            now=utcnow, name=obj.name.name,
            mean_sepal_length=obj.mean_sepal_length,
            mean_sepal_width=obj.mean_sepal_width,
            mean_petal_length=obj.mean_petal_length,
            mean_petal_width=obj.mean_petal_width)
```

This will generate a file like this:

```
$ cat statistics.log

        ALERT@2017-03-30 03:35:56.951190: Iris-setosa
            - mean_sepal_length = 5.006
            - mean_sepal_width  = 3.418
            - mean_petal_length = 1.464
            - mean_petal_width  = 0.244
        ------------------------------------------------------
        ALERT@2017-03-30 03:35:56.952553: Iris-versicolor
            - mean_sepal_length = 5.936
            - mean_sepal_width  = 2.77
            - mean_petal_length = 4.26
            - mean_petal_width  = 1.326
        ------------------------------------------------------
        ALERT@2017-03-30 03:35:56.954868: Iris-virginica
            - mean_sepal_length = 6.588
            - mean_sepal_width  = 2.974
            - mean_petal_length = 5.552
            - mean_petal_width  = 2.026
        ------------------------------------------------------
```

Email Endpoint

The `Email` endpoint takes a little bit more configuration.

First we need to configure the SMTP server (email server) in `settings.py`, like so

```
EMAIL = {
     "server": "smtp.foo.com:587",  # Host and port of SMTP server.
     "tls": True,  # If the smtp uses the TLS security
     "user": "foo@foo.com",  # User
     "password": "secret"  # Password
 }
```

Then when we add the endpoint to the alert, it is mandatory to add a list of destinations in the `to` parameter.

```
class StatisticsAlert(run.Alert):

    model = models.Statistics
    conditions = []
    alert_to = [ep.File("statistics.log"),
                ep.Email(to=["dest0@host.com", "dest1@host.com", ...])]
```

`Email` accepts three other optional parameters:

- `sent_from` a from email (by default we build one with the *user* and *host* of the SMTP configuration)
- `subject` a subject for the sent emails (default: name of the alert + name of the project)
- `message` a string that can have a slot to render the object, so that it can be used as a template to create the messages (it will use the method `render_alert()` of the alert by default.)

Selective Runs By Name and Groups

Just like the steps can be run by their names, Alerts can also be run this way by adding the parameter `--alerts|-a` to the `check-alerts` command. It is also possible to add alerts to groups with the attribute `groups` in Alert). We can selectively run this groups using the flag `--alert-groups|-ag`.

If you need more information, please check the tutorial for *Selective Steps Runs By Name and Groups*

4.2.6 Tutorial - Part #6 - Quality Assurance

This final section of the turorial are not necesary for write a Pipeline. Here we try to elavorate some concepts and tools to make you confident about your data reduction. So here you gonna find the most unique feature of all pipelene frameworks: An integrated Quality Assurance (QA) for make thrustworty pipelines.

Some words about QA

In "*Total Quality Control*" (Feigenbaum, 1983) Feigenbaum defines software quality as

> "Quality is a customer determination, not an engineer's determination, not a marketing determination, nor a general management determination. It is based on the customer's actual experience with the product or service, measured against his or her requirements – stated or unstated, conscious or merely sensed, technically operational or entirely subjective – and always representing a moving target in a competitive market"

In our context, a customer is not a single person but a role that our scientific requirements define, and the engineers are responsible for the design and development of a pipeline able to satisfy the functionality defined in those requirements. Measuring the quality of software is a task that involves the extraction of qualitative and quantitative metrics. One of the most common ways to measure Software Quality is Code Coverage (CC). To understand CC is necessary to define first the idea behind unit-testing. The unit-test objective is to isolate each part of the program and show that the individual parts are correct Jazayeri (2007). Following this, the CC is the ratio of code being executed by the tests –usually expressed as a percentage– (Miller and Maloney, 1963). Another metric in which we are interested, is the maintainability of the software. This may seem like a subjective parameter, but it can be measured using a standardization of code style; putting the number of style deviations as a tracer of code maintainability.

What is QA for Corral

As we said in the last paragraph, QA is subjective measure. That is the reason why Corral offers to the pipeline's author tools to deliver higher quality code. This tools can measure three quantities: - Unit-test results, measuring the expected functionality - Coverage, which stands for the amount of code being tested - Style, as a estimator of the mantainability

Corral offers three tools to generate a global status report that brings an idea about the pipeline's quality, so it is possible to share it to the stackholders.

Summarizing, is a pipeline's developer job to define: - Which are the minimum tests to check the pipeline's functionality - Assume that this testing set the baseline of the pipeline's quality - Assume the risks of deploying it's own code

Note: Following the subjectivity idea, this tool is optional, our original design comes from knowing ahead the amount of trust we put on deploying new versions of a pipeline, having settled before the "baseline"

Unit-Testing

From Wikipedia:

> Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class.
>
> Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when test cases fail, makes it less clear which component is causing the failure. Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher-quality unit that is also more maintainable.
>
> Wikipedia

In Corral's case, a sub-framework is offered, in which is proposed to test separatedly in one or many test, the loader, each step and each alert.

For instance if we would like to test if the **subject** `StatisticsCreator` each new instance of a `Statistics` for each instance of `Name`.

Note: We call **subject** to each step, loader and alert being put up to testing

```
from corral import qa

from . import models, steps

class StatisticsCreateAnyNameTest(qa.TestCase):

    subject = steps.StatisticsCreator

    def setup(self):
        name = models.Name(name="foo")
        self.save(name)

    def validate(self):
        self.assertStreamHas(
            models.Name, models.Name.name=="foo")
        self.assertStreamCount(1, models.Name)
```

(continues on next page)

(continued from previous page)

```
17
18          name = self.session.query(models.Name).first()
19
20          self.assertStreamHas(
21              models.Statistics, models.Statistics.name_id==name.id)
22          self.assertStreamCount(1, models.Statistics)
```

Breaking the code into pieces we have:

- On line number **5** we declare the test case, by setting a descriptive name and inhering from class `corral.qa.TestCase`.
- On line **7**, we link to the desired subject.
- From lines **9** and **11** (`setup()` method), we prepare and add to the data stream an instance of `Name` with any name, since we know from the step `StatisticCreator` definition that this model is being selected for an statistic.
- On `validate()` method (from line **13**) the data stream status after executing `StatisticCreator` is checked:
  - First of all on **14** and **15** lines it is verified that a effectively exists a `Name` instance in the stream with "foo" name.
  - In **16** it is checked that only one instance of `Name` exists on the stream (recall that each unit-test is executed isolated from every other, so whatever we added in `setup()` or whatever is being created by the **subject** are the only entities allowed to exist on the stream)
  - In line **18** we extract this one instance of `Name` from the stream
  - Finally on lines **20** - **22**, we verify that `StatisticsCreator` has created an instance of `Statistics` linked to the `Name` instance recently recovered, and that there is not any other instance in the Stream.

This testing example verifies the correct functioning of a simple step. Take into account that it is possible to create more than one test with each *subject*, by making variations on `setup()`, allowing to test different initialization parameters for *subject* and generalizing to each possible state.

Important: Take into account that a test is not **only** to check that the code works properly. In many cases it is key to check that the software *fails* just as it should.

For example if you code a Step that converts images, you probably want several tests taking into account the most common images, such as a properly formatted image, as well as an empty bytes string, or an image that cannot fit into memory.

Executing Tests

To run the previously descripted test the `test` command is used:

```
$ python in_corral.py test -vv
runTest (pipeline.tests.StatisticsCreateAnyNameTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.441s

OK
```

The `-vv` parameter increases the amount of information being screen printed. Now if we change the test, for instance the **16** line, and insert the following:

```
1  from corral import qa
2
3  from . import models, steps
4
5  class StatisticsCreateAnyNameTest(qa.TestCase):
6
7      subject = steps.StatisticsCreator
8
9      def setup(self):
10         name = models.Name(name="foo")
11         self.save(name)
12
13     def validate(self):
14         self.assertStreamHas(
15             models.Name, models.Name.name=="foo")
16         self.assertStreamCount(2, models.Name)
17
18         name = self.session.query(models.Name).first()
19
20         self.assertStreamHas(
21             models.Statistics, models.Statistics.name_id==name.id)
22         self.assertStreamCount(1, models.Statistics)
```

and execute `test` again, we should get the following:

```
$ python in_corral.py test -vv
runTest (pipeline.tests.StatisticsCreateAnyNameTest) ... FAIL

======================================================================
FAIL: runTest (pipeline.tests.StatisticsCreateAnyNameTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "corral/qa.py", line 171, in runTest
    self.validate()
  File "/irispl/tests.py", line 40, in validate
    self.assertStreamCount(2, models.Name)
  File "/corral/qa.py", line 251, in assertStreamCount
    self.assertEquals(query.count(), expected)
AssertionError: 1 != 2

----------------------------------------------------------------------
Ran 1 test in 0.445s

FAILED (failures=1)
```

This is due there are not 2 instances of `Name` in the Stream at that time.

Note: The `test` command supports a enormous quantity of parameters to activate or deactivate tests, depend its subject, or stopping the execution at the first error. Please execute `python in_corral test --help` to get every possible alternative

---

Mocks

In many situations it is compulsory to make use of certain Python functionalities (or another third party library), that exceeds subject's test scope, or any other kind of penalization with its use.

For example if we have any defined variable on `settings.py` called `DATA_PATH` which points where to store any processed file, and our subject creates data on that place. If we use this without caution our testing cases might get filled with trash files in our working directory.

Mock Objects might be useful in such times. These come already integrated inside `TestCase` from Corral, and their key advantage is that after getting out of the test case they are automatically whiped out.

```
import tempfile
import shutil

class SomeTest(qa.TestCase):

    subject = # some subject

    def setup(self):

        # create a temporary directory
        self.data_path = tempfile.tempdir()

        # change the settings.DATA_PATH and set it as our temporary directory
        self.patch("corral.conf.settings.DATA_PATH", self.data_path)

    def validate(self):
        # here, everything that makes use of DATA_PATH is being mocked

    def teardown(self):
        # here, everything that makes use of DATA_PATH is being mocked

        # clean the temporary file so we do not leave trash behind us
        shutil.rmtree(self.data_path)
```

The `teardown()` method does not need to restore `DATA_PATH` to its original value, we just use it (in that case) to set free disk space being utilized only inside the test.

Note: Corral mocks implement a big portion of Python mocks functionality, mainly de python, principalmente:

- `patch`
- `patch.object`
- `patch.dict`
- `patch.multiple`

For more information on how to use mocks pleas go to https://docs.python.org/3/library/unittest.mock.html

---

Corral Unit-Test Life cycle

Each unit-test is executed in isolation, to guarantee this Corral executes each of the following steps for **EACH** test case:

1. Every class which inherit from `corral.qa.TestCase` are collected in `tests.py` module

2. For each *TestCase* is being executed:
   (a) A testing database to contain the Stream is created.
   (b) Every model is created on the Stream.
   (c) A `session` is being created, to interact with the DB, and a test case is being assigned to it.
   (d) The `setup()` method is executed for the current testing case.
   (e) Database changes are confirmed and `session` is closed.
   (f) The `subject` is executed, and it comes with its own `session`.
   (g) A new `session` is created, and a testing case is assigned to it.
   (h) The `validate()` method is executed and `session` closes.
   (i) A new `session` is created and testing case is assigned.
   (j) The testing case's `teardown()` method is executed. This method is optional, and could be used for example to clean auxiliary files if needed.
   (k) The database is destroyed, and every mock is erased.
3. Results for each test are recovered.

Important: The fact of creating **4** different `session` to interact with the databases is guaranting that every communication inside the testing case is through the stream, and not through any other in-memory Python object.

Note: The default testing database is an in-memory SQLite (`"sqlite:///:memory:"`), but this can be overriden by setting the `TEST_CONNECTION` variable in the `settings.py` module

---

Code-Coverage

The unittest are a simple tool to check the correct functioning of the pipeline. To get an idea of how well are doing our tests we compute the Code-Coverage (CC), and is equal to the percentage of lines of code being executed in the tests.

Important: How important is Code-Coverage?

CC is of so important in quality, that has been included in:

- The guidelines by which avionics gear is certified by the Federal Aviation Administration is documented in DO-178B and DO-178C.
- is a requirement in part 6 of the automotive safety standard ISO 26262 Road Vehicles - Functional Safety.

Corral calculates CC as the ratio of lines executed in testing, with respect to the total number of code lines in the pipeline (also including tests).

Corral is capable of self calculating the CC in the quality report tool described below.

Code Style

The programming style (CS) is a set of rules or guidelines used when writing the source code for a computer program.

Python favours the legibility of code as a design idiosincracy, stablished on PEP20. The style guide which dictates beauty and legible code is presented on PEP8

CS it is often claimed that following a particular programming style will help programmers to read and understand source code conforming to the style, and help to avoid introducing errors.

In some ways CS is some kine of Maintainability

As in coverage CS is managed by Corral integrating the Flake 8 Tool and is informed inside the result of the reporting tool

Reporting

Corral is capable of generating a quality report over any pipeline with testing.

Corral inspects the code, documentation, and testing in order to infer a global view of the pipeline's quality and architecture.

To get access to this information we could use three commands.

1. `create-doc`

This command generates a Markdown version of an automatic manual for the pipeline, about Models, Loader, Steps, Alerts, and command line interface utilities, using the docstrings from the code itself.

When using the `-o` parameter we can switch the output to a file. In this case Corral will suggest render the information in 3 formats (HTML, LaTeX y PDF) using Pandoc (you will need to have Pandoc installed).

Example:

```
$ python in_corral.py create-doc -o doc.md
Your documentaton file 'doc.md' was created.

To convert your documentation to more suitable formats we sugest Pandoc
(http://pandoc.org/). Example:

 $ pandoc doc.md -o doc.html # HTML
 $ pandoc doc.md -o doc.tex  # LaTeX
 $ pandoc doc.md -o doc.pdf  # PDF via LaTeX
```

Output examples can be found at: https://github.com/toros-astro/corral/tree/master/docs/doc_output_examples

2. `create-models-diagram`

This creates a Class Diagram in Graphviz dot format.

When using the `-o` flag we can switch the output to a file. In this case Corral will attempt to render the diagram in a PNG using **Graphviz_** (you must install this library first).

```
$ python in_corral.py create-models-diagram -o models.dot
Your graph file 'models.dot' was created.

Render graph by graphviz:
 $ dot -Tpng models.dot > models.png

More Help: http://www.graphviz.org/
```

Examples of output diagrams in dot and PNG can be found at: https://github.com/toros-astro/corral/tree/master/docs/models_output_examples

3. qareport

Runs every test and Code Coverage evaluation, and uses this to create a Markdown document detailing the particular results of each testing stage, and finally calculates the QAI index outcome.

When using the `-o` parameter we can switch the output to a file. In this case Corral will suggest render the information in 3 formats (HTML, LaTeX y PDF) using Pandoc (you will need to have Pandoc installed).

```
$ python in_corral.py qareport -o report.md
[INFO] Running Test, Coverage and Style Check. Please Wait...
Your documentaton file 'report.md' was created.

To convert your documentation to more suitable formats we sugest Pandoc (http://
↪pandoc.org/). Example:
 $ pandoc report.md -o report.html # HTML
 $ pandoc report.md -o report.tex  # LaTeX
 $ pandoc report.md -o report.pdf  # PDF via LaTeX
```

Examples of reporting output available at: https://github.com/toros-astro/corral/tree/master/docs/qareport_output_examples

Notes about QAI (Quality Assurance Index)

We recognize the need of a value to quantify the pipeline software quality. For example, using different estimators for the stability and maintainability of the code, we arrived to the following Quality Index **includes in the QA Report**:

$$QAI = 2 \times \frac{TP \times \frac{T}{PN} \times COV}{1 + exp(\frac{MSE}{\tau \times PFN})}$$

The number of test passes and failures are the unit-testing results, that provide a reproducible and upda-table manner to decide whether your code is working as expected or not. The *TP* factor is a critical feature of the index, since it is discrete, and if a single unit test fails it sets the QAI to zero, displaying that if your own tests fail then no result is guaranteed to be reproducible.

The $\frac{T}{PN}$ factor is a measure of how many of the different processing stages critical to the pipeline are being tested (a low value on this parameter should be interpreted as a need to write new tests for each pipeline stage).

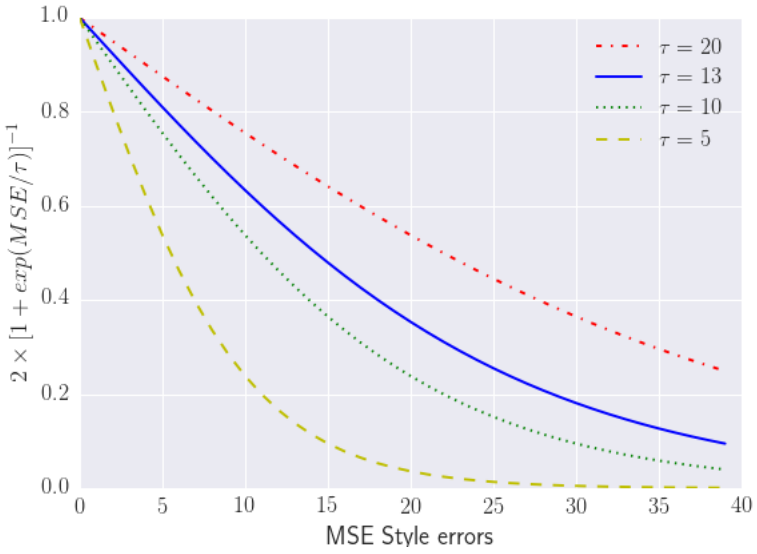
The COV factor shows the percentage of code that is being executed in the sum of every unit test; this displays the "quality of the testing" (a low value should be interpreted as a need to write more extensive tests).

The last factor is the one involving the exponential of the $\frac{MSE}{\tau}$ value. It comprises the information regarding style errors, attenuated by a default or a user-defined tolerance τ times the number of files in the project PFN The factor 2 is a normalization constant, so that $QAI \in [0, 1]$.

Note: By default $\tau = 13$ (the number of style errors on a single python script) is empirically determined from a random sample of more than 4000 python scripts.

You can change it by defining a variable on `settings.py` called `QAI_TAU` and asigned some number to it.

As you can see in the graph the slope (penalization) of the QAI curve is lower when τ is bigger.



Notes about QA Qualification

The QA Qualification (QAQ) is a quantitave scale based on QAI. Its a single symbol asigned to some range of a QAI to decide is your code aproves or not your expeted level of confidence. By default the top limits of the QAQ are the same system used by three different colleges in the United States:

- Dutchess Community College
- Carleton College
- Wellesley College

Where

- If your $QAI >= 0.00$
- If your $QAI >= 60.00$
- If your $QAI >= 63.00$
- If your $QAI >= 67.00$
- If your $QAI >= 70.00$
- If your $QAI >= 73.00$
- If your $QAI >= 77.00$
- If your $QAI >= 80.00$
- If your $QAI >= 83.00$
- If your $QAI >= 87.00$
- If your $QAI >= 90.00$
- If your $QAI >= 93.00$
- If your $QAI >= 95.00$

This values are defined by a dictionary in the form

```
{
    0: "F",
    60: "D-",
    63: "D",
    67: "D+",
    70: "C-",
    73: "C",
    77: "C+",
    80: "B-",
    83: "B",
    87: "B+",
    90: "A-",
    93: "A",
    95: "A+"
}
```

As you can see every key is the lower limit of the QAQ, you can change this by adding the `SCORE_CUALIFITACIONS` variable to the `settings.py` of your pipeline.

For example if you want to simple send a "**fail**" or "**pass**" message when your pipeline QAI are below or under 60

```
SCORE_CUALIFICATIONS = {
    0: "FAIL",
    60: "PASS"
}
```

See also:

If you're new to Python, you might want to start by getting an idea of what the language is like. Corral is 100% Python, so if you've got minimal comfort with Python you'll probably get a lot more out of Corral.

If you're new to programming entirely, you might want to start with this list of Python resources for non-programmers

If you already know a few other languages and want to get up to speed with Python quickly, we recommend Dive Into Python. If that's not quite your style, there are many other books about Python.

4.3 Topics

Introductions to all the key parts of Corral you'll need to know:

Contents:

4.3.1 The command line interface

The Corral library gives you the power to manage a chain of processes, or pipeline, that relies on a database by delivering command line commands.

This works for example for creating a databased:

```
$python in_corral.py createdb
$python in_corral.py sqlitebrowser
```

And if you have the sqlitebrowser program installed you should be able to open in a window a database manager and search into the contents of your data structures.

Another feature of corral is the ability to execute a shell environment where you have the most important imports already done, giving you even a `session` instance from sqlalchemy working and ready to receive queries and to commit entries.

This can be done by simply typing `python in_corral.py shell` in your terminal, and it will simply give you a IPython shell, or if you don't have IPython a bPython -in case you also lack of a bPython interpreter a plain python prompt is what you get-.

Even more Corral can give you a IPython Notebook by running

```
$ python in_corral.py notebook
```

Other useful utility is the `exec` command available, which can give a script for input to the Corral environment, just as if you were importing the script on a shell. I works by running:

```
$ python in_corral.py exec your_script.py
```

4.3.2 Integrate Corral with Django

Django is...

> ... a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source. [Read Mode]

So this chapter will teach how to access Corral managed database from django.

Lets asume we are trying to integrate a pipeline called *my_pipeline*

First for isolate every view inside a SqlAlchemy transaction add to your middleware list `corral.libs.django_integration.CorralSessionMiddleware`

Finally edit your `settings.py` file and add to the end of the code.

```
os.environ.setdefault("CORRAL_SETTINGS_MODULE", "my_pipeline.settings")

from corral import core
core.setup_environment()
```

Now you can use all the functionaly of corral from python and access a SQLALchemy session from every request.

Example

```
# cmodels to avoid django models name
from my_pypeline import models as cmodels

def django_view(request):
    session = request.corral_session
    session.query(MyModel).filter(MyModel.attr=="Foo")
    ...
```

Also if you want to exclude a view from the Corral scope you can add the decorator `@no_corral`

Example

```
from corral.lib.django_integration import no_corral

@no_corral
def django_view(request):
    ...
```

4.4 Glossary

pipeline A Python package – i.e. a directory of code – that contains all the settings for an instance of Corral. This would include database configuration, Corral-specific options and application-specific settings. [Read More]

MVC Model–view–controller (MVC) is a software design pattern for implementing user interfaces on computers. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. [Read More]

Index