# Flock Documentation

***Release 1.0.0***

**Captalys Data Science Team**

**Dec 04, 2017**

# Contents

# CHAPTER 1

# Installation

In order to use Flock Multiprocessing you only need to install it with pip:

```
pip install flockmp
```

## 1.1 Contribute

Please, we want to hear back from you. Go to the github page and make your contributions too. Your help can range from ideas and improve documentation to actually coding new features!

If you want to clone/fork the repository, follow these steps.

Follow the next steps in order to install the `Flock` module:

1. `git clone http://github.com/Captalys/Flock`
2. `cd Flock`
3. `python setup.py develop`
4. `python -m unittest discover test`

Make sure that all the tests run correctly.

# CHAPTER 2

# Getting Started

- Go to the *Installation* page.
- **Learn the basic concepts of Flock.**
    - *`FunctionAsync`*
    - *`DataFrameAsync`*
    - *`ListAsync`*
    - `DatabaseAsync`
- **Dive in to other tutorials below.**
    - *Basic Function*
    - *Lambdas*
    - *Instance methods*
    - *DataFrames*
    - *Database dependent functions*

## 2.1 Basic Function

We only need to define our iterator, which are the elements that will be applied to the given function. After that, we use the `apply()` from the *`FunctionAsync`*.

```
def myFunction(value):
    tmp = value ** 2
    return tmp

iterator = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
res = FunctionAsync.apply(iterator, myFunction)
res>
     [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

## 2.2 Lambdas

Usage is the same if you have a `lambda()` function.

```
iterator = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
res = FunctionAsync.apply(iterator, lambda x: x ** 2)
res>
     [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

## 2.3 Instance methods

The regular `multiprocessing` module can't handle instance methods very well because they are not *picklable* objects. However, you can continue to use the same interface as before for instance methods.

```
class Test(object):
    def compute(self, foo, bar):
        tmp = foo + bar
        res = tmp ** foo
        return res


inst = Test()
iterator = [(val1, val2) for val1, val2 in zip([1,2,3,4], [10,20,30,40])]
res = FunctionAsync(iterator, inst.compute)
res>
     [11, 44, 99, 176]
```

## 2.4 DataFrames

There are two use cases related to DataFrames already implemented. First, you might want to execute the `apply()` function in a row-by-row basis. For example, in order to create a new column using two existent columns.

```
df = DataFrame({"foo": [5, 10, 15, 20],
                "bar": [1, 2, 3, 4]})
df["new-var"] = DataFrameAsync.apply(df[["foo", "bar"]],
                                     lambda x: (x["foo"] + x["bar"]) ** 2, style="row-
↪like")
```

Using the previous method, `Flock` will split your DataFrame into chunks, send each chunk to a specific process, and inside each process it will multiprocess each row. This approach is very scalable if you have a very large dataframe and only want to perform an apply method.

The next use case is block based. Imagine you want to use your entire dataframe as input to some operation that will be applied to every column.

```
df = DataFrame({"foo": list(range(1000)),
                "bar": list(range(2000, 3000))})
df_new = DataFrameAsync.apply(df, lambda x: x ** 2, style="block-like")
```

## 2.5 Database dependent functions

This is a very useful class if you work with many databases in your code base. One of the main problems with multiprocessing and databases is related to the impossibility of sending a *connection* object to each open process. This becomes very annoying since you have multiple databases with several drivers.

One known solution to this problem is the guidance to open your connection inside the multiprocessed function. However, this is a very bad idea sometimes because the time you might take to connect can be very long and you will not gain the full benefits of multiprocessing.

The strategy adopted by `Flock` is to divide this problem into two steps. First, you need to create a `DatabaseSetup` instance to inform all the connections and name variables you are using inside the function you desire to multiprocess.

Using this instance, `Flock` will establish all your needed connections only once per process and reuse the connections for each task that processes get assigned to perform. Let's see that in action mocking a MySQL connection (SQLAlchemy) and a Apache Cassandra connection (cassandra-driver).

```
def myFunction(value, cass_con, mysql_con):
    getData = pd.read_sql("select * from cool_table", mysql_con)
    saveData = cass_con.execute("insert data to your cassandra_cool table")
    return True


# you probably have a method to connect to the database. Send the method without␣
↪making the call
mysqlCreateCon = MysqlConnectionClass.yourConnectMethod
cassCreateCon = CassandraConnectionClass.yourConnectMethod


# create the setup instances
mysqlSetup = DatabaseSetup(server=mysqlCreateCon, name="mysql_con",
                           parameters={"password": "password1", "user": "root"})

cassSetup = DatabaseSetup(server=cassCreateCon, name="cass_con",
                          parameters={"keyspace": "__default__", "ip": "0.0.0.0"})

# now we send the two setup connections to the databaseasync
dbAsync = DatabaseAsync(setups=[mysqlSetup, cassSetup])
res = dbAsync.apply(iterator=[1, 2, 3, 4, 5, 6], myFunction)
```

In the setup process, the attribute *name* should be the same value as the *variable name* inside the `myFunction()` that will be processed. As you can see, the setup process can be a little boring, so we have a `BaseDatabaseSetup` to be extended and you can hide all this portion inside your code.

CHAPTER 3

# Multiprocessing your **Functions**

**Flock supports the following list of functions:**

- Lambdas
- Instance methods
- Class methods
- Regular functions

If your function needs a connection to any database to perform its computation, please use the `DatabaseAsync` class.

**class** `flockmp.`**`FunctionAsync`**

`FunctionAsync` is a class to apply and manage multiprocessing tasks within functions.

**classmethod** **`apply`**(*iterator*, *function*, *poolSize=5*)

Method `apply()` executes a function asynchronously given an iterator.

**Parameters**

- **`iterator`** (*`iter`*) – variable in which the function will be applied.
- **`function`** (*`func`*) – a function in which is desired to be ran multi-processed.

Returns the list with the results of function(iterator).

## 3.1 Example

```
_list = list(range(2000))
res = FunctionAsync.apply(_list, lambda x: x ** 2 / 10)
```

CHAPTER 4

# Multiprocessing **Database** dependent functions

## 4.1 `DatabaseSetup`

Abstract class used to extend and create your custom interfaces to connect the desired database.

# CHAPTER 5

# Multiprocessing **DataFrame** objects

**class** `flockmp.dataframe.`**`DataFrameAsync`**

**classmethod** **`apply`** (*dataframe*, *function*, *style='row-like'*, *chunksize=100*, *poolSize=5*)

First we segmentat the orginal `DataFrame` in chunks, then the `executeAsync()` will parallelize the function's operations on the segmented dataframes. There two options for the way it will operate, as *row-like* or *block-like*.

**Parameters**

- **`dataframe`** (*`DataFrame`*) – Input Dataframe
- **`fuction`** (*`func`*) – Function to be applied on the dataframe
- **`chunksize`** (*`int`*) – How many chunks the original dataframe will be splitted
- **`poolSize`** (*`int`*) – Number of pools of processes
- **`style`** (*`str`*) – if "row-like" `function()` will be applied in row-by-row, otherwise it will be applied in `DataFrame` chunks.

## 5.1 Example

```
df = DataFrame({"a": list(range(1000)),
                "b": list(range(1000, 2000))})
res = DataFrameAsync.apply(df, lambda x: x ** 2, style="block-like")
```

# CHAPTER 6

# Multiprocessing **List** objects

**class** `flockmp.list.`**`ListAsync`**

**classmethod** **`apply`**(*_list*, *function*, *poolSize=5*)

First we segmentat the orginal `List` in chunks, then the `executeAsync()` will parallelize the function's operations on the segmented lists.

**Parameters**

- **`_list`** (*`list`*) – Input List
- **`fuction`** (*`func`*) – Function to be applied on the list
- **`poolSize`** (*`int`*) – Number of pools of processes

## 6.1 Example

```
_list = list(range(2000))
res = ListAsync.apply(_list, lambda x: x ** 2 / 10)
```

# CHAPTER 7

# Flock Base class

Base class used to build the `ListAsync`, `DataFrameAsync` and `FunctionAsync` classes.

You should avoid using this class inside your code. The interface might change without previous notice.

**class** `flockmp.base.`**`BaseMultiProc`**(*poolSize=5*, *timeOutError=50*, *context='spawn'*)

`BaseMultiProc` is a class to apply and manage multiprocessing tasks within functions.

**Parameters**

- **`poolSize`** (`int`) – amount of resources set to be processed the same time (*default* = 5)
- **`timeOutError`** (`int`) – degree of tolerance for waiting a process to end (*default* = 50)
- **`context`** (`str`) – way of starting a process (depends os the platform). It can be "spawn", "fork" or "forkserver" ('default'="spawn")

**`executeAsync`**(*function*, *iterator*)

Method `executeAsync()` executes a function asynchronally, given a set of iterators.

**Parameters**

- **`function`** (`func`) – a function in which is desired to be ran multiprocessed and asynchronally
- **`iterator`** (`iter`) – variable in wich the function will be applied

Returns the result of the function given a set of arguments of that function.

## 7.1 Example

```
bp = BaseMultiProc()
iterator = list(range(10))
res = bp.executeAsync(lambda x: x ** 2, iterator)
```

# CHAPTER 8

# Indices and tables

- genindex
- modindex
- search

# Index