sparklanes Documentation

Kevin Baumgarten

Mar 09, 2020

Contents:

sparklanes is a lightweight data processing framework for Apache Spark. It was built with the intention to make building complex spark processing pipelines simpler, by shifting the focus towards writing data processing code without having to spent much time on the surrounding application architecture.

Data processing pipelines, or *lanes*, are built by stringing together encapsulated processor classes, which allows creation of lane definitions with an arbitrary processor order, where processors can be easily removed, added or swapped.

Processing pipelines can be defined using *lane configuration YAML files*, to then be packaged and submitted to spark using a single command. Alternatively, the same can be achieved manually by using the framework's API.

CHAPTER 1

Writing Data Processing Tasks

1.1 Creating a Task

In sparklanes, data processing tasks exist as decorated classes. Best practice suggests, that a task should depend as little as possible on other tasks, in order to allow for lane definitions with an arbitrary processor order (up to a certain extent, because of course there will always be some dependence, since e.g. a task extracting data most likely comes before one transforming it).

For example:

```
from sparklanes import Task

@Task('extract_data')
class ExtractIrisCSVData(object):
    def __init__(self, iris_csv_path):
        self.iris_csv_path = iris_csv_path

    def extract_data(self):
        # Read the csv
        iris_df = conn.spark.read.csv(path=self.iris_csv_path,
                                      sep=',',
                                      header=True,
                                      inferSchema=True)
```

The class `ExtractIrisCSVData` above becomes a *Task* by being decorated with *`sparklanes.Task()`*. Tasks have an *entry*-method, which is the method that will be run during lane execution, and is specified using the `Task` decorator's sole argument (in this case, `extract_data`).

The entry-method itself should not take arguments, however custom arguments can be passed to the class during instantiation.

Todo: The functionality to pass args/kwargs to both the constructor, as well as to the entry method, might be added in future versions.

1.2 Sharing Resources between Tasks

By being decorated, the class becomes a child of the internal *sparklanes._framework.task.LaneTask* class and inherits the *sparklanes._framework.task.LaneTask.cache()*, *sparklanes._framework.task.LaneTask.uncache()* and *sparklanes._framework.task.LaneTask.clear_cache()* methods, which can be used to add an object to the *TaskCache*.

When object is cached from within a task (e.g. using `self.cache('some_df', df)`, it becomes an attribute to all tasks that follow and is accessible from within each task object as `self.some_df` (that is, until it is uncached).

1.3 Accessing the Pyspark API from within Tasks

To allow for easy access to the Pyspark API across tasks, sparklanes offers means to avoid having to "getOrCreate" a `SparkContext`/`SparkSession` in each task requiring access to one. A module containing tasks can simply import *sparklanes.conn* (an alias of *sparklanes.SparkContextAndSessionContainer*) and have immediate access to a SparkContext and SparkSession object:

```
from sparklanes import conn

conn.sc     # SparkContext instance
conn.spark  # SparkSession instance
```

The currently active Context/Session can be changed using its methods *sparklanes.SparkContextAndSessionContainer.set_sc()* and *sparklanes.SparkContextAndSessionContainer.set_spark()*

If it is preferred to handle SparkContexts/SparkSessions manually, without making use of the shared container, this can be done by setting an environment variable `INIT_SPARK_ON_IMPORT` to `0` when submitting the application to spark.

CHAPTER 2

Defining Processing Lanes

Lanes (i.e. data processing pipelines) can be defined by either writing YAML definition files, or by using sparklane's API.

2.1 YAML definition files

Lane definition files must adhere to the following schema:

```yaml
lane:
  name: SimpleLane          # str (optional): Name under which the lane will be␣
↪referred to during logging
  run_parallel: false       # bool (optional): Indicates whether tasks should be run␣
↪in parallel
  tasks:                    # list (required): List of processor classes
    - class: pkg.mdl.Task1  # str (required): Full python module path to the␣
↪processor class
      args: [arg1, arg2]    # list (optional): List of arguments passed when␣
↪instantiating the class
      kwargs:               # dict (optional): Dict of kwargs passed when␣
↪instantiating the class
        kwarg1: val1
        kwarg2: val2
    - class: pkg.mdl.Task2
      args: [arg1, arg2]
    ...
```

Attention should be placed on using the correct class path. Let's say we have the following directory structure:

```
tasks/
  extract/
    __init__.py
    extractors.py    # Contains class 'SomeExtractorClass'
  load/
```

(continues on next page)

(continued from previous page)

```
    ...
...
```

The exact path to be used then depends on which folder will be packaged and submitted to spark. To reference `SomeExtractorClass`, the correct class path would be `tasks.extract.extractors.SomeExtractorClass` if the entire `tasks` folder would be packaged and submitted to Spark, whereas just packaging the `extract` folder would result in the correct class path `extract.extractors.SomeExtractorClass` (see *Submitting lanes to Spark*).

2.2 Using the API

Lanes can also be defined and executed using sparklane's API, for example:

```
from sparklanes import Lane, Task

@Task('main_mtd')
class Task1(object):
    def main_mtd(self, a, b, c):
        pass

@Task('main_mtd')
class Task2(object):
    def main_mtd(self, a, b):
        pass

# Building the lane
lane = (Lane(name='ExampleLane', run_parallel=False)
        .add(Task1, 1, 2, c=3)
        .add(Task2, a=1, b=2))

# Execute it
lane.run()
```

Refer to the API documentation for *sparklanes.Lane*.

2.3 Branching & Running Tasks in Parallel

Lanes can be branched infinitely deep, which is especially useful if part of the lane should be executed in parallel. As stated in the Spark documentation:

```
Inside a given Spark application (SparkContext instance), multiple parallel jobs can␣
↪run
simultaneously if they were submitted from separate threads.
```

If parameter `run_parallel` is true when instantiating a `Lane` or `Branch`, a separate thread will be spawned for each of the tasks it contains, ensuring that Spark will execute them in parallel.

For example, a lane containing branches could look like this:

```
from sparklanes import Lane, Branch
from pkg.mdl import Task1, Task2, Task3, SubTaskA, SubTaskB1, SubTaskB2, SubTaskC
```

(continues on next page)

(continued from previous page)

```
lane = (Lane(name='BranchedLane', run_parallel=False)
        .add(Task1)
        .add(Task2)
        .add(Branch(name='ExampleBranch', run_parallel=True)
             .add(SubTaskA)
             .add(Branch(name='SubBranch', run_parallel=False)
                  .add(SubTaskB1)
                  .add(SubTaskB2))
             .add(SubTaskC))
        .add(Task3))
```

Or the same lane defined as YAML:

```
lane:
  name: BranchedLane
  run_parallel: false
  tasks:
    - class: pkg.mdl.Task1
    - class: pkg.mdl.Task2
    - branch:
      name: ExampleBranch
      run_parallel: true
      tasks:
        - class: pkg.mdl.SubTaskA
        - branch:
          name: ExampleSubBranch
          run_parallel: false
          tasks:
          - class: pkg.mdl.SubTaskB1
          - class: pkg.mdl.SubTaskB2
        - class: pkg.mdl.SubTaskC
```

In this lane, `SubTaskA`, Branch `SubBranch` and `SubTaskC` would be executed in parallel, whereas the tasks within `SubBranch` wouldn't be. This way, complex processing pipelines can be built.

Refer to *sparklanes.Branch*.

CHAPTER 3

Submitting lanes to Spark

3.1 Console script

sparklanes comes with a console script to package and submit a YAML lane to Spark:

```
usage: lane-submit [-h] -y YAML -p PACKAGE [-r REQUIREMENTS]
                   [-e [EXTRA_DATA [EXTRA_DATA ...]]] [-m MAIN]
                   [-d SPARK_HOME] [-s [SPARK_ARGS [SPARK_ARGS ...]]]
                   [--silent]

Submitting a lane to spark.

optional arguments:
  -h, --help            show this help message and exit
  -y YAML, --yaml YAML  Path to the yaml definition file.
  -p PACKAGE, --package PACKAGE
                        Path to the python package containing your tasks.
  -r REQUIREMENTS, --requirements REQUIREMENTS
                        Path to a `requirements.txt` specifying any additional
                        dependencies of your tasks.
  -e [EXTRA_DATA [EXTRA_DATA ...]], --extra-data [EXTRA_DATA [EXTRA_DATA ...]]
                        Path to any additional files or directories that
                        should be packaged and sent to Spark.
  -m MAIN, --main MAIN  Path to a custom main python file
  -d SPARK_HOME, --spark-home SPARK_HOME
                        Custom path to the directory containing your Spark
                        installation. If none is given, sparklanes will try to
                        use the `spark-submit` command from your PATH
  -s [SPARK_ARGS [SPARK_ARGS ...]], --spark-args [SPARK_ARGS [SPARK_ARGS ...]]
                        Any additional arguments that should be sent to Spark
                        via spark-submit. (e.g. `--spark-args executor-
                        memory=20G total-executor-cores=100`)
  --silent              If set, no output will be sent to console
```

3.2 Packaging

When submitting a YAML lane configuration file to spark, the python package containing the tasks (i.e. the data processors) has to be specified. While there is no strict requirement anymore for python packages to have a `__init__.py` for Python version `3.4+`, it remains a requirement here.

For example, if a YAML file contains tasks like:

```
tasks:
- class: tasks.extract.LoadFromS3
- class: tasks.extract.LoadFromMySQL
- class: tasks.transform.NormalizeColumns
- class: tasks.transform.EngineerFeatures
- class: tasks.load.DumpToFTP
```

Then the directory structure of the python package specified using `-p` should look something like this:

```
tasks/
  __init__.py
  extract.py  # Contains LoadFromS3 and LoadFromMySQL classes
  transform.py  # Contains NormalizeColumns and EngineerFeatures classes
  load.py  # Contains DumpToFTP class
```

3.3 Extra Data

If any additional data needs to be accessible locally from within the spark cluster, they can be specified using `-e/--extra-data`. Both files and directories are supported, and they will be accessible relative to the application's root directory.

For example, a single file, as in `-e example.csv`, will be made accessible from spark at `./example.csv`, regardless from the original directory structure. If a directory is specified, e.g. `-e extra/data`, that folder will be accessible from spark at `./data`.

3.4 Spark Configuration

Any flags and configuration arguments accepted by spark-submit can also be used using `lane-submit`.

For example, `spark-submit` configuration arguments could look like:

```
spark-submit --properties-file ./spark.conf --executor-memory 20G --supervise [...]
```

Then those same arguments could be passed using `lane-submit` like:

```
lane-submit -s properties-file=./spark.conf executor-memory=20G supervise [...]
```

3.5 Custom main

The default main python file is a simple script loading and executing the lane:

```
"""A simple 'main' file, that will be submitted to spark. It will execute the lane as␣
↪defined in
the YAML lane definition file."""
# pylint: disable=missing-docstring
from argparse import ArgumentParser

from sparklanes import build_lane_from_yaml


def main():
    args = parse_args()
    build_lane_from_yaml(args['lane']).run()


def parse_args():
    parser = ArgumentParser()
    parser.add_argument('-l', '--lane',
                        help='Relative or absolute path to the lane definition YAML␣
↪file',
                        type=str,
                        required=True)

    return parser.parse_args().__dict__


if __name__ == '__main__':
    main()
```

If this is not sufficient, the script can be extended and the new python script specified as a custom main file as in `-m new_main.py`.

CHAPTER 4

API reference

4.1 sparklanes

class sparklanes.**Branch**(*name='UnnamedBranch'*, *run_parallel=False*)
Bases: *sparklanes.Lane*, object

Branches can be used to split task lanes into branches, which is e.g. useful if part of the data processing pipeline should be executed in parallel, while other parts should be run in subsequent order.

class sparklanes.**Lane**(*name='UnnamedLane'*, *run_parallel=False*)
Bases: object

Used to build and run data processing lanes (i.e. pipelines). Public methods are chainable.

add(*cls_or_branch*, **args*, ***kwargs*)
Adds a task or branch to the lane.

Parameters

- **cls_or_branch** (*Class*) –
- ***args** – Variable length argument list to be passed to *cls_or_branch* during instantiation
- ****kwargs** – Variable length keyword arguments to be passed to *cls_or_branch* during instantiation

Returns self

Return type Returns *self* to allow method chaining

run()
Executes the tasks in the lane in the order in which they have been added, unless *self.run_parallel* is True, then a thread is spawned for each task and executed in parallel (note that task threads are still spawned in the order in which they were added).

class sparklanes.**SparkContextAndSessionContainer**
Bases: object

Container class holding SparkContext and SparkSession instances, so that any changes will be propagated across the application

classmethod init_default()
Create and initialize a default SparkContext and SparkSession

sc = None

classmethod set_sc(*master=None*, *appName=None*, *sparkHome=None*, *pyFiles=None*, *environment=None*, *batchSize=0*, *serializer=PickleSerializer()*, *conf=None*, *gateway=None*, *jsc=None*, *profiler_cls=<class 'pyspark.profiler.BasicProfiler'>*)
Creates and initializes a new *SparkContext* (the old one will be stopped). Argument signature is copied from pyspark.SparkContext.

classmethod set_spark(*master=None*, *appName=None*, *conf=None*, *hive_support=False*)
Creates and initializes a new *SparkSession*. Argument signature is copied from pyspark.sql.SparkSession.

spark = None

sparklanes.**Task**(*entry*)
Decorator with which classes, who act as tasks in a *Lane*, must be decorated. When a class is being decorated, it becomes a child of *LaneTask*.

Parameters **entry** (*The name of the task's "main" method, i.e. the method which is executed when task is run*) –

Returns **wrapper (function)**

Return type The actual decorator function

sparklanes.**build_lane_from_yaml**(*path*)
Builds a *sparklanes.Lane* object from a YAML definition file.

Parameters **path** (*str*) – Path to the YAML definition file

Returns Lane, built according to definition in YAML file

Return type *Lane*

sparklanes.**conn**
alias of *sparklanes.SparkContextAndSessionContainer*

4.2 Internals

4.2.1 sparklanes._framework package

sparklanes._framework.env module

Environment configuration variables that can be passed when executing a lane.

sparklanes._framework.env.**INIT_SPARK_ON_IMPORT = False**
Specifies if a default SparkContext/SparkSession should be instantiated upon import sparklanes

sparklanes._framework.env.**INTERNAL_LOGGER_NAME = 'SPARKLANES'**
The logger's name under which internal events will be logged.

sparklanes._framework.env.**SPARK_APP_NAME = 'sparklanes.app'**
The app name using which the default SparkContext/SparkSession will be instantiated

sparklanes._framework.env.**UNNAMED_BRANCH_NAME = 'UnnamedBranch'**
Default name of a branch, if no custom branch is specified

sparklanes._framework.env.**UNNAMED_LANE_NAME = 'UnnamedLane'**
Default name of a lane, if no custom name is specified.

sparklanes._framework.env.**VERBOSE_TESTING = False**
Specifies if output should be printed to console when running tests

sparklanes._framework.errors module

Exceptions

exception sparklanes._framework.errors.**CacheError**
Bases: AttributeError

Should be thrown whenever task-cache access fails.

exception sparklanes._framework.errors.**LaneExecutionError**
Bases: Exception

Should be thrown whenever execution of a lane fails.

exception sparklanes._framework.errors.**LaneImportError**
Bases: Exception

Should be thrown when a module or class in a YAML definition file cannot be imported.

exception sparklanes._framework.errors.**LaneSchemaError**(**args*, ***kwargs*)
Bases: schema.SchemaError

Should be thrown when a YAML definition does not match the required schema.

code
Show the specific error from the super class

exception sparklanes._framework.errors.**TaskInitializationError**
Bases: Exception

Should be thrown whenever transformation of a class into a task fails (during decoration).

sparklanes._framework.lane module

Lane and Branch classes. TODO: Better logging

sparklanes._framework.log module

Module handling logging. TODO: improve logging. Allow configuration, etc.

sparklanes._framework.log.**make_default_logger**(*name='SPARKLANES'*, *level=20*, *fmt='%(asctime)s - %(name)s - %(levelname)s - %(message)s'*)
Create a logger with the default configuration

sparklanes._framework.spark module

Used to allow sharing of SparkContext and SparkSession, to avoid having to "getOrCreate" them again and again for each task. This way, they can just be imported and used right away.

sparklanes._framework.task module

Includes the *Task* decorator, the parent class *LaneTask* from which all tasks will inherit, as well as the *_TaskCache*, which is used to share attributes between Task objects.

class sparklanes._framework.task.**LaneTask**
Bases: object

The super class of each task, from which all tasks inherit when being decorated with *sparklanes.Task*

cache(*name*, *val*, *overwrite=True*)
Assigns an attribute reference to all subsequent tasks. For example, if a task caches a DataFrame *df* using *self.cache('some_df', df)*, all tasks that follow can access the DataFrame using *self.some_df*. Note that manually assigned attributes that share the same name have precedence over cached attributes.

Parameters

- **name** (*str*) – Name of the attribute
- **val** – Attribute value
- **overwrite** (*bool*) – Indicates if the attribute shall be overwritten, or not (if *False*, and a cached attribute with the given name already exists, *sparklanes.errors.CacheError* will be thrown).

clear_cache()
Clears the entire cache

uncache(*name*)
Removes an attribute from the cache, i.e. it will be deleted and becomes unavailable for all subsequent tasks.

Parameters **name** (*str*) – Name of the cached attribute, which shall be deleted

class sparklanes._framework.task.**LaneTaskThread**(*task*)
Bases: threading.Thread

Used to spawn tasks as threads to be run in parallel.

join(*timeout=None*)
Overwrites *threading.Thread.join*, to allow handling of exceptions thrown by threads from within the main app.

run()
Overwrites *threading.Thread.run*, to allow handling of exceptions thrown by threads from within the main app.

class sparklanes._framework.task.**TaskCache**
Bases: object

Serves as the attribute cache of tasks, which is accessed using the tasks' *__getattr__* method.

cached = {}

static get(*name*)
Retrieves an object from the cache.

Parameters **name** (*str*) – Name of the object to be retrieved

Returns

Return type object

sparklanes._framework.validation module

Contains helper functions, used for class and schema validation.

sparklanes._framework.validation.**arg_spec**(*cls*, *mtd_name*)
Cross-version argument signature inspection

Parameters

- **cls** (*class*) –
- **mtd_name** (*str*) – Name of the method to be inspected

Returns

- **required_params** (*list of str*) – List of required, positional parameters
- **optional_params** (*list of str*) – List of optional parameters, i.e. parameters with a default value

sparklanes._framework.validation.**validate_params**(*cls*, *mtd_name*, **args*, ***kwargs*)
Validates if the given args/kwargs match the method signature. Checks if: - at least all required args/kwargs are given - no redundant args/kwargs are given

Parameters

- **cls** (*Class*) –
- **mtd_name** (*str*) – Name of the method whose parameters shall be validated
- **args** (*list*) – Positional arguments
- **kwargs** (*dict*) – Dict of keyword arguments

sparklanes._framework.validation.**validate_schema**(*yaml_def*, *branch=False*)
Validates the schema of a dict

Parameters

- **yaml_def** (*dict*) – dict whose schema shall be validated
- **branch** (*bool*) – Indicates whether *yaml_def* is a dict of a top-level lane, or of a branch inside a lane (needed for recursion)

Returns True if validation was successful

Return type bool

4.2.2 sparklanes._submit package

sparklanes._submit._main module

A simple 'main' file, that will be submitted to spark. It will execute the lane as defined in the YAML lane definition file.

sparklanes._submit._main.**main**()

sparklanes._submit._main.**parse_args**()

sparklanes._submit.submit module

Module that allows submitting lanes to spark using YAML definitions

sparklanes._submit.submit.**submit_to_spark**()
Console-script entry point

CHAPTER 5

Example: Simple ETL lane

Note: it is recommended to took a look at the 'official documentation <https://sparklanes.readthedocs.io>'__ first.

In this example, we'll build a simple ETL pipeline using sparklanes and the popular iris dataset. First, we'll load the dataset from CSV, before applying some transformations on it, to then finally dump it as JSON to disk.

Let's start by importing all the libs we need:

[6]:
```
from pyspark.sql.functions import monotonically_increasing_id

from sparklanes import Task, Lane, conn
```

And write our data processors, or `Tasks` next:

[6]:
```
@Task('extract_data')
class ExtractIrisCSVData(object):
    """Load the iris data set from a CSV file"""
    def __init__(self, iris_csv_path):
        self.iris_csv_path = iris_csv_path

    def extract_data(self):
        # Read the csv
        iris_df = conn.spark.read.csv(path=self.iris_csv_path,
                                      sep=',',
                                      header=True,
                                      inferSchema=True)

        # Make it available to tasks that follow
        self.cache('iris_df', iris_df)


@Task('add_index')
class AddRowIndex(object):
    """Add a index to each row in the data set"""
    def add_index(self):
        # Add id column
```

(continues on next page)

(continued from previous page)

```
        self.iris_df = self.iris_df.withColumn('id', monotonically_increasing_id())

        # Update cache
        self.cache('iris_df', self.iris_df)


@Task('normalize')
class NormalizeColumns(object):
    """Normalize all numerical columns"""
    def normalize(self):
        # Add normalized columns
        columns = self.iris_df.columns
        columns.remove('species')
        for col in columns:
            col_min = float(self.iris_df.agg({col: "min"}).collect()[0]['min(%s)' %␣
↪col])
            col_max = float(self.iris_df.agg({col: "max"}).collect()[0]['max(%s)' %␣
↪col])
            self.iris_df = self.iris_df.withColumn(
                col + '_norm', (self.iris_df[col] - col_min) / (col_max - col_min)
            )

        # Update Cache
        self.cache('iris_df', self.iris_df)


@Task('write_to_json')
class SaveAsJSON(object):
    """Dump the data set as JSON to disk"""
    def __init__(self, output_folder):
        self.output_folder = output_folder

    def write_to_json(self):
        self.iris_df.write.format('json').save(self.output_folder)

        # Clear cache
        self.uncache('iris_df')
```

Note how in the extractor class, we cache our `DataFrame` using `self.cache`, and as a result make it an attribute to all tasks that follow.

With our Tasks being defined, we can now build the lane:

[11]:
```
lane = (Lane(name='IrisExampleLane', run_parallel=False)
        .add(ExtractIrisCSVData, iris_csv_path='data/iris.csv')
        .add(AddRowIndex)
        .add(NormalizeColumns)
        .add(SaveAsJSON, 'out'))
```

And run it:

[12]:
```
lane.run()
```

```
2018-06-07 11:52:01,466 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing `IrisExampleLane`
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
2018-06-07 11:52:01,468 - SPARKLANES - INFO -
================================================================================
        IrisExampleLane
         >Task_ExtractIrisCSVData
         >Task_AddRowIndex
         >Task_NormalizeColumns
         >Task_SaveAsJSON
================================================================================
2018-06-07 11:52:01,469 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `ExtractIrisCSVData.extract_data`
--------------------------------------------------------------------------------
2018-06-07 11:52:01,668 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `ExtractIrisCSVData.extract_data`. Execution time: 0:00:00.
↪198397
--------------------------------------------------------------------------------
2018-06-07 11:52:01,669 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `AddRowIndex.add_index`
--------------------------------------------------------------------------------
2018-06-07 11:52:01,675 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `AddRowIndex.add_index`. Execution time: 0:00:00.004853
--------------------------------------------------------------------------------
2018-06-07 11:52:01,676 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `NormalizeColumns.normalize`
--------------------------------------------------------------------------------
2018-06-07 11:52:02,848 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `NormalizeColumns.normalize`. Execution time: 0:00:01.170293
--------------------------------------------------------------------------------
2018-06-07 11:52:02,849 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `SaveAsJSON.write_to_json`
--------------------------------------------------------------------------------
2018-06-07 11:52:03,065 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `SaveAsJSON.write_to_json`. Execution time: 0:00:00.215517
--------------------------------------------------------------------------------
2018-06-07 11:52:03,066 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing `IrisExampleLane`
--------------------------------------------------------------------------------
```

[12]:
```
<sparklanes._framework.lane.Lane at 0x1034bf5f8>
```

That works, but what makes `sparklanes` more useful, is the capability of defining processing lanes using *YAML configuration files*, to then submit these lanes to a spark cluster.

We can define the same lane as above like:

```yaml
lane:
  name: IrisExampleLane
  run_parallel: false
```

(continues on next page)

(continued from previous page)

```
  tasks:
    - class: tasks.iris.ExtractIrisCSVData
      kwargs:
        iris_csv_path: data/iris.csv
    - class: tasks.iris.AddRowIndex
    - class: tasks.iris.NormalizeColumns
    - class: tasks.iris.SaveAsJSON
      args:
        - out
```

With the file being saved as `iris.yml`, our directory structure looks like this:

```
data/
  iris.csv
tasks/
  __init__.py  # Required to be recognized as a python package
  iris.py  # Contains our processor classes (Tasks)
iris.yml
```

So to run our pipeline, we can submit it to spark using the `lane-submit` command line script:

[10]:
```
%%bash
lane-submit -y iris.yml -p tasks -e data -s master=local[2]
```

```
2018-06-07 11:50:09,219 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing `IrisExampleLane`
--------------------------------------------------------------------------------

2018-06-07 11:50:09,219 - SPARKLANES - INFO -
================================================================================
        IrisExampleLane
         >Task_ExtractIrisCSVData
         >Task_AddRowIndex
         >Task_NormalizeColumns
         >Task_SaveAsJSON
================================================================================
2018-06-07 11:50:09,219 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `ExtractIrisCSVData.extract_data`
--------------------------------------------------------------------------------
2018-06-07 11:50:13,784 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `ExtractIrisCSVData.extract_data`. Execution time: 0:00:04.
↪564362
--------------------------------------------------------------------------------
2018-06-07 11:50:13,784 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `AddRowIndex.add_index`
--------------------------------------------------------------------------------
2018-06-07 11:50:13,818 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `AddRowIndex.add_index`. Execution time: 0:00:00.033710
--------------------------------------------------------------------------------
2018-06-07 11:50:13,818 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `NormalizeColumns.normalize`
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------
2018-06-07 11:50:16,216 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `NormalizeColumns.normalize`. Execution time: 0:00:02.398010
--------------------------------------------------------------------------------
2018-06-07 11:50:16,216 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Executing task `SaveAsJSON.write_to_json`
--------------------------------------------------------------------------------
2018-06-07 11:50:16,833 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing task `SaveAsJSON.write_to_json`. Execution time: 0:00:00.616633
--------------------------------------------------------------------------------
2018-06-07 11:50:16,833 - SPARKLANES - INFO -
--------------------------------------------------------------------------------
Finished executing `IrisExampleLane`
--------------------------------------------------------------------------------
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use␣
↪setLogLevel(newLevel).
```

That's it. Please check out the documentation for explanations on how more complex processing lanes can be built.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

T

U

V