pyperunner

Release 0.1.2

Gregor Lichtner

Sep 02, 2021

CONTENTS:

1 PypeRunner - Pure Python ETL Pipeline **1**
- 1.1 Features . . . 1
- 1.2 Installation . . . 1
- 1.3 Quickstart . . . 2
- 1.4 Documentation . . . 5
- 1.5 Examples . . . 5

2 Tutorial **7**
- 2.1 Example scripts . . . 7
- 2.2 Define work tasks . . . 7
- 2.3 Combine tasks in pipeline . . . 7
- 2.4 Run the pipeline . . . 7
- 2.5 Modify parameters and rerun . . . 7
- 2.6 Modify pipeline and rerun . . . 7
- 2.7 Access the results . . . 7

3 PypeRunner Components **9**
- 3.1 Task . . . 9
- 3.2 Pipeline . . . 10
- 3.3 Runner . . . 10

4 Define Tasks **11**
- 4.1 Function as task . . . 11
- 4.2 Class as task . . . 12

5 Combine Tasks to Pipeline **15**
- 5.1 Standard Pipeline . . . 15
- 5.2 Sequential Pipeline . . . 15
- 5.3 Pipeline summary . . . 16
- 5.4 Save & load pipelines . . . 17
- 5.5 Run Pipeline . . . 18

6 Access pipeline results **19**
- 6.1 pipeline.results . . . 19
- 6.2 PipelineResult . . . 20
- 6.3 Filesystem . . . 20

7 Reproducibility **21**

8 Result Caching **23**
- 8.1 Examples . . . 23

9 API **25**
9.1 Overview . . . 25
9.2 Task . . . 25
9.3 Pipeline . . . 29
9.4 Runner . . . 30
9.5 PipelineResult . . . 34

10 Authors **37**

11 Change Log **39**
11.1 Version History . . . 39

12 Indices and tables **41**

Index **43**

CHAPTER

ONE

PYPERUNNER - PURE PYTHON ETL PIPELINE

PypeRunner is an easy to use yet powerful workflow pipeline tool written in pure python, with parallel processing of tasks, smart caching of results and reproducible runs. PypeRunner allows for the creation of complex workflows with branching and merging of several concurrent execution flows.

1.1 Features

- Easy to use extract, transform load (ETL) pipeline tool
- Reproducible pipeline runs for data science / machine learning tasks
- Easy creation of pipelines using functional API chaining (see below)
- Parallel processing of steps
- Caching of previously run steps to speed up processing
- Re-run of steps (and all subsequent steps) when parameters are changed
- Save & read pipelines to/from yaml
- Graphical output of pipeline run

1.2 Installation

Install from conda forge via

```
conda install pyperunner -c conda-forge
```

Install from pip via

```
pip install pyperunner
```

Or from source via

```
git clone https://github.com/glichtner/pyperunner.git
cd pyperunner
python setup.py install
```

1.3 Quickstart

Pyperunner has three basic components:

- **Task**: Definition of the work to do (Python classes or functions)
- **Pipeline**: A collection of tasks that are connected in a directed fashion
- **Runner**: The executor of a pipeline

1.3.1 Hello world example

```
from pyperunner import Runner, Pipeline, task


@task("Hello", receives_input=False)
def hello():
    print("in hello()")
    return "Hello"


@task("World")
def world(data):
    hello = data["Hello()"]
    print("in world()")
    return f"{hello} world"


# instantiate and connect tasks
hello = hello()
world = world()(hello)

# create pipeline and set root element
pipeline = Pipeline("hello-world-example", [hello])

# print a summary of the pipeline
pipeline.summary()

# run pipeline
runner = Runner(data_path="data/", log_path="log/")
runner.run(pipeline)

# get pipeline results object from the pipeline that was just run
results = runner.results()

# show the results
```

(continues on next page)

(continued from previous page)

```
for task_name in results:
    print(f"Output of task '{task_name}' was '{results[task_name]}'")
```

Running this script outputs the following:

```
~/pyperunner/examples$ python hello-world-func.py

+---------+
| Hello() |
+---------+
      *
      *
      *
+---------+
| World() |
+---------+

2021-01-03 20:55:47 INFO     MainProcess  root       Storing pipeline parameters in␣
↪examples/log/hello-world-example_210103T205547/pipeline.yaml
2021-01-03 20:55:47 INFO     MainProcess  root       Storing pipeline data in examples/
↪data
2021-01-03 20:55:47 INFO     Process-1    Hello()    Starting
2021-01-03 20:55:47 INFO     Process-1    Hello()    in hello()
2021-01-03 20:55:47 INFO     Process-1    Hello()    Finished: Status.SUCCESS
2021-01-03 20:55:47 INFO     Process-2    World()    Starting
2021-01-03 20:55:47 INFO     Process-2    World()    in world()
2021-01-03 20:55:47 INFO     Process-2    World()    Finished: Status.SUCCESS
2021-01-03 20:55:47 INFO     MainProcess  root       Pipeline run finished

Output of task 'Hello()' was 'Hello'
Output of task 'World()' was 'Hello world'
```

Note that if you re-run the script, pyperunner will detect that the current configuration has already run and will skip the execution of these tasks:

```
~/pyperunner/examples$ python hello-world.py

2021-01-03 20:56:36 INFO     MainProcess  root       No need to execute task "Hello()",␣
↪skipping it
2021-01-03 20:56:36 INFO     MainProcess  root       No need to execute task "World()",␣
↪skipping it
2021-01-03 20:56:36 INFO     MainProcess  root       Pipeline run finished
```

If you need to reprocess outputs, just add the *force_reload=True* parameter to the pipeline run:

```
runner.run(pipeline, force_reload=True)
```

Or to run just a specific task again, use the *reload=True* parameter when initializing the task:

```
# instantiate and connect tasks
hello = hello()
world = world(reload=True)(hello)
```

Note that pyperunner detects which tasks it must re-execute: All depending tasks of a reloaded task are automatically

re-executed, and only those tasks are fully skipped from execution from which the output is not required in a successor task. Also, if a task has been previously executed and its output is required, it is read from disk.

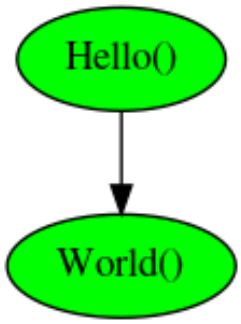
```
~/pyperunner/examples$ python hello-world.py

2021-01-03 20:57:26 INFO     Process-1    Hello()    Starting
2021-01-03 20:57:26 INFO     Process-1    Hello()    Loading output from disk, skipping␣
↪processing
2021-01-03 20:57:26 INFO     Process-1    Hello()    Finished: Status.SUCCESS
2021-01-03 20:57:26 INFO     Process-2    World()    Starting
2021-01-03 20:57:26 INFO     Process-2    World()    in world()
2021-01-03 20:57:26 INFO     Process-2    World()    Finished: Status.SUCCESS
2021-01-03 20:57:26 INFO     MainProcess  root       Pipeline run finished
```

At each run, the pipeline is automatically stored in a yaml file in the log path to ensure reproducibility:

```yaml
pipeline:
  name: hello-world-example
tasks:
  Hello():
    hash: 22179f3afd85ab64dd32c63bc21a9eb4
    module: __main__
    name: Hello
    params: {}
    parents: []
    tag: ''
  World():
    hash: f7d904856f2aa4fda20e05521298397f
    module: __main__
    name: World
    params: {}
    parents:
    - Hello()
    tag: ''
```

Additionally, a graphical representation of the run is saved in the log path:



1.4 Documentation

The API Reference provides API-level documentation.

1.5 Examples

Look in the examples/ directory for some example scripts.

1.5.1 Multiple paths pipeline

```
# Create pipeline
pipeline = Pipeline("my-pipeline")

# Create first stream of tasks: LoadData(csv) --> ProcessData(normalize-l2)
load_db = LoadData(
    "database",
    database={"host": "localhost", "username": "user", "password": "password"},
    wait=10,
)
norm_l2 = ProcessData("normalize-l2", norm="l2", axis=0, wait=1)(load_db)

# Create second stream of tasks:
#  LoadData(csv) --> ProcessData(normalize-l1) --> AugmentData(augment)
load_csv = LoadData("csv", filename="data.csv", wait=1)
norm_l1 = ProcessData("normalize-l1", norm="l1", wait=1)(load_csv)
augment = AugmentData("augment", types=["rotate", "noise"], wait=1)(norm_l1)

# Combine outputs of both streams (ProcessData(normalize-l2)
# and AugmentData(augment)), additionally add output from ProcessData(normalize-l1)
evaluate = Evaluate("both", wait=1)([norm_l1, norm_l2, augment])

# Add the roots of both streams to the pipeline
pipeline.add(load_db)
pipeline.add(load_csv)

# print a summary of the pipeline
pipeline.summary()

# Run pipeline
runner = Runner(data_path="data/", log_path="log/")
runner.run(pipeline, force_reload=False)
```

pipeline.summary() prints the following ascii summary:

```
                                                                  +---------------+
                                                                  | LoadData(csv) |
                                                                  +---------------+
                                                                          *
                                                                          *
                                                                          *
```

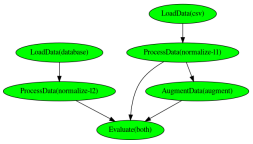
(continues on next page)

(continued from previous page)

```
    +---------------------+                                    +-------------------------
↪---+
    | LoadData(database) |                                    | ProcessData(normalize-
↪l1) |
    +---------------------+                                    +-------------------------
↪---+
              *                                                   ***                  ␣
↪***
              *                                                ****                    ␣
↪   ***
              *                                             **                         ␣
↪      ****
+----------------------------+                 +-----------------------+               ␣
↪       ****
| ProcessData(normalize-l2) |                 | AugmentData(augment) |                ␣
↪*******
+----------------------------+****             +-----------------------+           *******
                                 *******                    *                 *******
                                        *******             *           *******
                                               ****         *       ****
                                                 +-----------------+
                                                 | Evaluate(both) |
                                                 +-----------------+
```

Notice how multiple tasks run simultaneously:

```
2021-01-03 19:09:05 INFO     Process-1    LoadData(csv)                 Starting
2021-01-03 19:09:05 INFO     Process-2    LoadData(database)            Starting
2021-01-03 19:09:06 INFO     Process-1    LoadData(csv)                 Finished:␣
↪Status.SUCCESS
2021-01-03 19:09:06 INFO     Process-3    ProcessData(normalize-l1)     Starting
2021-01-03 19:09:07 INFO     Process-3    ProcessData(normalize-l1)     Finished:␣
↪Status.SUCCESS
2021-01-03 19:09:07 INFO     Process-4    AugmentData(augment)          Starting
2021-01-03 19:09:08 INFO     Process-4    AugmentData(augment)          Finished:␣
↪Status.SUCCESS
2021-01-03 19:09:15 INFO     Process-2    LoadData(database)            Finished:␣
↪Status.SUCCESS
2021-01-03 19:09:15 INFO     Process-5    ProcessData(normalize-l2)     Starting
2021-01-03 19:09:16 INFO     Process-5    ProcessData(normalize-l2)     Finished:␣
↪Status.SUCCESS
2021-01-03 19:09:16 INFO     Process-6    Evaluate(both)                Starting
2021-01-03 19:09:17 INFO     Process-6    Evaluate(both)                Finished:␣
↪Status.SUCCESS
```



CHAPTER

TWO

TUTORIAL

2.1 Example scripts

A collection in example scripts that are ready to run can be found in the examples folder.

Script Name	Description
hello-world-class.py	Tasks implemented as classes using the `run()` decorator
hello-world-func.py	Tasks implemented as functions using the `task()` decorator
multi-path.py	Complex pipeline with multiple branches

2.2 Define work tasks

2.3 Combine tasks in pipeline

2.4 Run the pipeline

2.5 Modify parameters and rerun

2.6 Modify pipeline and rerun

2.7 Access the results

CHAPTER

THREE

PYPERUNNER COMPONENTS

PypeRunner has three main building blocks:

- *Task* define the work units
- *Pipeline* defines the order in which the tasks are executed
- *Runner* orchestrates the execution of tasks in a pipeline

3.1 Task

A task is the

3.1.1 Task Name

The name of a task consists of its class name and an optional tag in the form of *<class-name>(<tag>)*. As the name of each task in a pipeline must be unique, the tag can be used to use multiple instances of the same task in a pipeline.

```
@task('simple-task')
def simpletask(data):
    return 'simple-task'

first_task = simpletask(tag='first')
second_task = simpletask(tag='second')

print(first_task.name)
# returns 'simple-task(first)'
print(second_task.name)
# return 'simple-task(sceond)'
```

3.1.2 Task Parameter

A task that receives data from an upstream task (i.e. all tasks that are not at the root / start of a pipeline) needs to accept a *data* parameter, in which the upstream tasks' results will be provided to the task. The input data is provided in a list, where each element of the list contains the result of one upstream task. For example, if a task has three upstream (parent) tasks, then the *data* argument to the task's function will contain three items, one from each upstream task. Please note that the order of elements in *data* is not guaranteed and the upstream tasks need to supply identifying information in their result on their own, should that be required to process their outputs in the downstream child task.

A task can have additional parameters. These need to be defined as the task's function (either in the function's definition, if defining the task by a function, or in *run()*, if defining a task as a class) and need to be specified during task creation time. Additionally

3.1.3 Task Hash

The task hash is an unique identifier of a task that takes into account both the task's parameter and the task's environment in the pipeline (specifically: ALL upstream tasks). It is computed using the *hash()* method.

The hash is constructed from the hash of the specific task and the hashes of *all* parent (predecessor) tasks. It is therefore dependent on the pipeline the task is part of. This is used to precisely identify a task in a given context (e.g. when saving and loading pipelines using the *pyperunner.Pipeline.to_file()* and *pyperunner.Pipeline.from_file()* methods to store/load a pipeline from file). It ensures reproducibility of single pipeline runs.

3.1.4 Memory consumption

Note that pyperunner executes each task in a separate process (using the python library multiprocessing) and receives the results using a multiprocessing.Queue. Thus, memory consumption may be more than twice the size of actual task return value as it must be stored both in the task and in the main process that receives the data.

3.2 Pipeline

A *Pipeline* describes the connections between tasks.

3.3 Runner

A *Runner* performs and orchestrates the actual execution of tasks. Pyperunner executes each task in a separate process and can therefore inherently parallelize tasks.

CHAPTER

FOUR

DEFINE TASKS

Tasks can be defined either from functions or by subclassing the *Task* class, Which way to use depends on the complexity and organization of the task's code: For a simple workflow a function usually suffices and is easier readable, while for complex workflows with multiple function calls and the requirement to keep a state, a class might be the better choice.

4.1 Function as task

To use a user-defined function as a task, simply tag it with the `task()` decorator:

```
from pyperunner import task

@task("MyTask")
def my_function(data):
    result = do_something(data)
    return result

# note that creating the task is performed by calling the function
# but *without* the data argument.
my_task = my_function()
```

The `task()` decorator has a required positional parameter *name*, which is used as the name of the task. It is required that each task in pipeline has a unique name.

The function that is decorated by the `task()` decorator needs to either accept a named parameter *data* or the task() decorator needs to be supplied a *receives_input=False* parameter.

```
# Create a task named "SimpleTask" - note that the function accepts
# the `data` keyword parameter
@task('SimpleTaskWithInput')
def simpletask_with_input(data):
    return 'simple-task'

# If the function should not accept data (i.e. be a starting task of the pipeline),
# the `receives_input=False` parameter must be supplied to the task() decorator.
@task('SimpleTaskWithoutInput', receives_input=False)
def simpletask_without_input():
    return 'simple-task'

# The following definition will raise an AttributeError
```

(continues on next page)

(continued from previous page)

```
@task('SimpleTaskError')
def simpletask_without_input_error():
    return 'simple-task'
# raises AttributeError: To receive input data, the function must accept
# the named parameter "data"
```

You can add additional parameters to the function definition. These then need to be supplied during task creation time:

```
@task("MyTaskWithParameters")
def my_function(data, reduce, n_iterations=10):
    result = data
    for i in range(n_iterations):
        result = do_something(result, reduce=reduce)

    return result

# The task is created by calling the function but always *without*
# the `data` argument.
my_task = my_function(reduce=True, n_iterations=20)

# Parameters with default values may be skipped during task creation
my_task = my_function(reduce=True)
```

4.2 Class as task

To use a class as a task, the class must inherit from the *Task* class, implement the abstract *run()* function and use the `run()` decorator on the *run()* implementation:

```
from pyperunner import Task, run

class Hello(Task):
    @run
    def run(self, data):
        return "Hello"


# Note that creating the task is performed by instantiating the class:
my_task = Hello()
```

You can add additional parameters to the *run()* method definition. These then need to be supplied during task creation time:

```
@task("MyTaskWithParameters")
class SomeWorker(Task):
    @run
    def run(self, data, reduce, n_iterations=10):
        result = data
        for i in range(n_iterations):
            result = do_something(result, reduce=reduce)
```

(continues on next page)

(continued from previous page)

```
        return result

# The task is created by instantiating the class using the additional
# parameters from the run function as parameters to the constructor
# but always *without* the `data` argument.
my_task = SomeWorker(reduce=True, n_iterations=20)

# Parameters with default values may be skipped during task creation
my_task = SomeWorker(reduce=True)
```

CHAPTER

FIVE

COMBINE TASKS TO PIPELINE

5.1 Standard Pipeline

A *Pipeline* describes the connections between tasks. Create a pipeline using:

```
from pyperunner import Pipeline

pipeline = Pipeline("my-pipeline")
```

Pyperunner offers a functional API to connect tasks and to add them to a pipeline:

```
@task('simple-task')
def simpletask(data):
    return 'simple-task'

# Reusing the same task using the "tag" parameter
first_task = simpletask(tag='first')
second_task = simpletask(tag='second')
third_task = simpletask(tag='third')
fourth_task = simpletask(tag='fourth')

# connect tasks from first to fourth task
fourth_task(third_task(second_task(first_task)))

# and add the starting task to the pipeline
pipeline.add(first_task)
```

5.2 Sequential Pipeline

A *Sequential* pipeline is a wrapper for purely sequential (path) workflows. With sequential pipelines, you can supply the whole pipeline as a list to the constructor. The tasks are automatically connected in the supplied order and applied

```
@task('simple-task')
def simpletask(data):
    return 'simple-task'

# Reusing the same task using the "tag" parameter
first_task = simpletask(tag='first')
second_task = simpletask(tag='second')
```

(continues on next page)

(continued from previous page)

```
third_task = simpletask(tag='third')
fourth_task = simpletask(tag='fourth')

# create a sequential pipeline and supply whole pipeline as list in constructor
pipeline = Sequential("sequential", [first_task, second_task, third_task, fourth_task])
```

5.3 Pipeline summary

Pyperunner offers two modes of visualizing the pipeline: As ASCII and as PNG:

```
# create pipeline and add tasks
# ...

# print an ASCII summary
pipeline.summary()
```

This outputs an ASCII summary like this to the console:

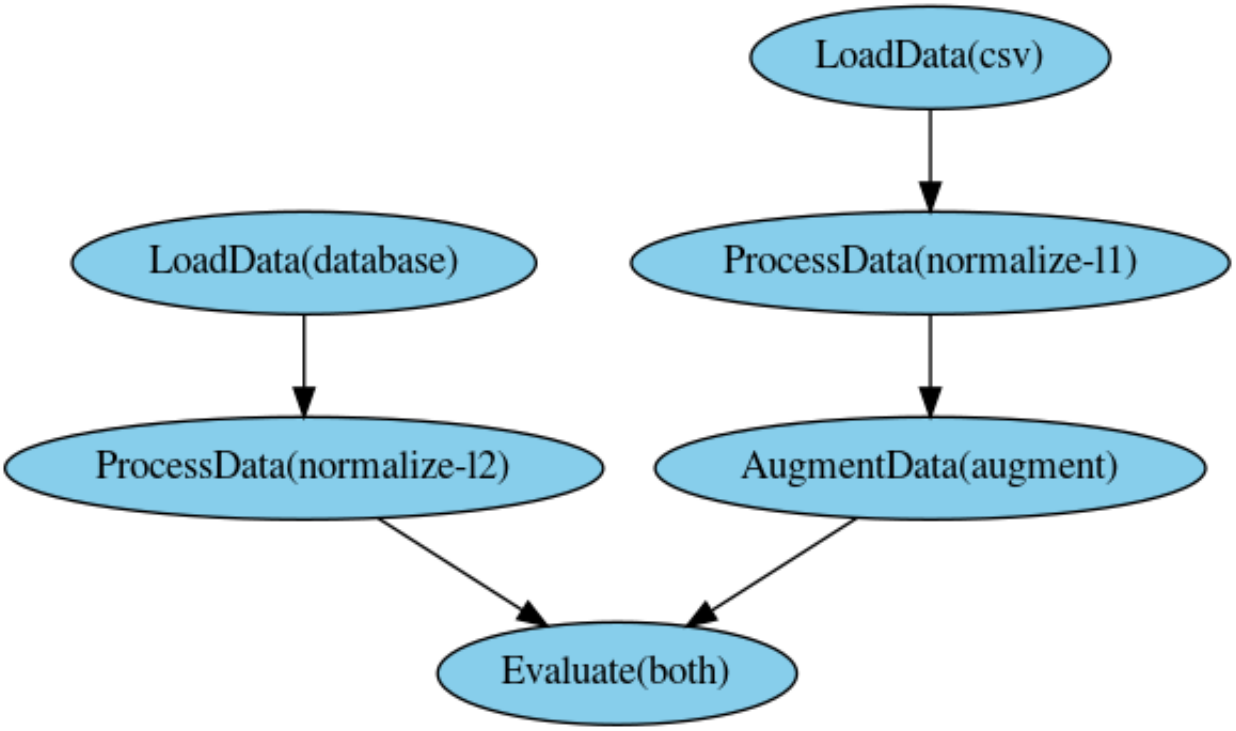
```
                                                +---------------+
                                                | LoadData(csv) |
                                                +---------------+
                                                        *
                                                        *
                                                        *
    +--------------------+                +-----------------------------+
    | LoadData(database) |                | ProcessData(normalize-l1) |
    +--------------------+                +-----------------------------+
              *                                         *
              *                                         *
              *                                         *
+----------------------------+                  +-----------------------+
| ProcessData(normalize-l2) |                  | AugmentData(augment) |
+----------------------------+                  +-----------------------+
                         ****                     ***
                             ****           ****
                                 **       **
                            +-----------------+
                            | Evaluate(both) |
                            +-----------------+
```

The PNG image of the pipeline is created by the *Runner*, because it contains color-coded information about the pipeline run

```
# Run pipeline
runner = Runner(data_path="data/", log_path="log/")
runner.run(pipeline)

runner.write_status_image("status.img")
```

This outputs an image like this:



5.4 Save & load pipelines

At each run, information about the pipeline is automatically saved to the log path as *pipeline.yaml*. To manually save a pipeline, call the *to_file()* function of the pipeline

```
pipeline.to_file('my-pipeline.yml')
```

To load and fully reconstruct a previously saved pipeline, use the *from_file()* method:

```
from pyperunner import Pipeline, Runner

pipeline = Pipeline.from_file('my-pipeline.yml')

# re-run loaded pipeline
runner = Runner(data_path='data/', log_path='log/')
runner.run(pipeline)
```

Note: To reconstruct a pipeline from yaml file, you must make sure that all Tasks used by the pipeline are defined/imported when calling *from_file()* and that the defined Tasks use the exact same modules and names as saved in the pipeline. This is because the pipeline tasks and their modules are stored as strings in the yaml file and only reconstructed from their names.

5.5 Run Pipeline

To run a pipeline, instantiate a *Runner* and run the pipeline. You need to specify the data and log paths for the runner, where data_path is the path where all outputs of the tasks are stored and log_path is the path where log file from the actual pipeline run, along the pipeline definition as a yaml file, are stored.

```
runner = Runner(data_path='data/', log_path='log/')
runner.run(pipeline)
```

CHAPTER

SIX

ACCESS PIPELINE RESULTS

You have three options to access the pipeline results: 1. Using the `results()` function after a pipeline run 2. Using the *pyperunner.PipelineResult.from_file()* function on a stored pipeline run yaml file 3. Manually from the file system using joblib.load

6.1 pipeline.results

After a pipeline run, you can access the results using `results()` on the pipeline:

```
runner = Runner(data_path='data/', log_path='log/')
runner.run(pipeline)

list(pipeline.results())

# outputs, for example:
# [
#   'AugmentData(augment)',
#   'Evaluate(both)',
#   'LoadData(csv)',
#   'LoadData(database)',
#   'ProcessData(normalize-l1)',
#   'ProcessData(normalize-l2)'
# ]
```

You can access the results from individual tasks using the task name as the key to the *pyperunner.PipelineResult()* object:

```
for task_name in pipeline.results():
    print(pipeline[task_name])
```

6.2 PipelineResult

To access the results of a pipeline at a later timepoint, use the *pyperunner.PipelineResult.from_file()* function on a stored pipeline run yaml file.

```
runner = Runner(data_path='data/', log_path='log/')
runner.run(pipeline)
```

Example output:

```
2021-01-23 17:23:18 INFO     MainProcess  root      Stored pipeline parameters in /home/
↪glichtner/projects/pyperunner/log/my-pipeline_210123T172317/pipeline.yaml
```

Load the pipeline results:

```
from pyperunner import PipelineResult
fname = "/home/glichtner/projects/pyperunner/log/my-pipeline_210123T172317/pipeline.yaml"

results = PipelineResult.from_file(fname)

# loop through individual task results
for task_name in pipeline.results():
    print(pipeline[task_name])
```

6.3 Filesystem

The outputs of tasks are dumped to the local filesystem using joblib.dump(). To load the output of a single task manually, use joblib.load():

```
import joblib
fname = "/home/glichtner/projects/pyperunner/data/Evaluate(both)/
↪197f82a6dbc9799406a35bef412cd7f4/result.dump.gz"

task_result = joblib.load(fname)
```

CHAPTER

SEVEN

REPRODUCIBILITY

To ensure reproducibility, alongside the results of each task result, pyperunner stores the paramters used for that task and also a hash of the parent/upstream tasks of that task.

CHAPTER

EIGHT

RESULT CACHING

Pyperunner caches the outputs of a task and only executes a task if

- The task hasn't been executed yet and there are no results available
- The task is forced to be executed by specifying *reload=True* during task creation
- All tasks are forced to be executed by using *force_reload=True* when calling *pyperunner.Runner.run()*
- Different parameters are used when instantiating the task
- Any task upstream to the task in question has changed or is required to reload

8.1 Examples

Consider the following Sequential pipeline of four tasks:

```
@task('simple-task')
def simpletask(data, param):
    return 'simple-task'

# Reusing the same task using the "tag" parameter
first_task = simpletask(tag='first', param=1)
second_task = simpletask(tag='second', param=2)
third_task = simpletask(tag='third', param=3)
fourth_task = simpletask(tag='fourth', param=4)

# create a sequential pipeline and supply whole pipeline as list in constructor
pipeline = Sequential("sequential", [first_task, second_task, third_task, fourth_task])

runner = Runner(data_path='data/', log_path='log/')
runner.run(pipeline)
```

If these task haven't run before in the specified data_path, all 4 tasks are executed. If running the pipeline directly again, no task will be executed, as all results have been cached in data_path and nothing has changed:

```
runner.run(pipeline) # no task executed, all cached
```

If a task is recreated using the *reload* paramter, that task and all subsequent tasks are executed:

```
third_task = simpletask(tag='third', param=3, reload=True)

# create a sequential pipeline and supply whole pipeline as list in constructor
```

(continues on next page)

(continued from previous page)

```
pipeline = Sequential("sequential", [first_task, second_task, third_task, fourth_task])

runner.run(pipeline) # executes task 3 (because of reload=True) and 4 (because its␣
↪upstream task changed)
```

All tasks can be executed by using the force_reload parameter on the runner:

```
# create a sequential pipeline and supply whole pipeline as list in constructor
pipeline = Sequential("sequential", [first_task, second_task, third_task, fourth_task])

runner.run(pipeline, force_reload=True) # executes all tasks
```

If any parameter of task is changed (and the task hasn't been executed using that parameter configuration), then also that task and all dependent (downstream) tasks are executed:

```
# Reusing the same task using the "tag" parameter
first_task = simpletask(tag='first', param=1)
second_task = simpletask(tag='second', param='CHANGED') # this task uses a different␣
↪parameter value
third_task = simpletask(tag='third', param=3)
fourth_task = simpletask(tag='fourth', param=4)

# create a sequential pipeline and supply whole pipeline as list in constructor
pipeline = Sequential("sequential", [first_task, second_task, third_task, fourth_task])

runner.run(pipeline) # Executes task 2 (because of the changed parameter) and tasks 3␣
↪and 4
                     # (because an upstream task is executed)
```

CHAPTER

NINE

API

9.1 Overview

This section describes pyperunner's API.

Task([tag, reload])	A task is a single work unit that is run as part of a *Pipeline*.
Runner.run([pipeline, force_reload, ...])	Run a pipeline.
Pipeline.from_file(filename[, compare_hashes])	Create a pipeline from a stored file yaml file.
PipelineResult(conf[, data_path])	Accessor of task results of a pipeline run.

9.2 Task

class pyperunner.**Task**(*tag=''*, *reload=False*, ***kwargs*)

A task is a single work unit that is run as part of a *Pipeline*.

All tasks that are run by a pipeline must subclass this class. There are two main ways to accomplish this:

1. Using the `task()` function decorator:

```
from pyperunner import task

@task("Hello")
def hello():
    print("in hello()")
    return "Hello"
```

Note that in this case you need to explicitly state the name of the task as a parameter to the `task()` function decorator (here: "Hello").

2. Directly subclassing this class and then using the `run()` method decorator on the run() function. Note that the abstract *run()* function must be implemented when subclassing.

```
from pyperunner import run

class World(Task):
    @run
    def run(self, data):
        return f"{data} world"
```

Note that in contrast to the `task()` function decorator, you don't need to specify the task name here. Instead, the class name (here "World") will be used as the task name.

Parameters

- **tag** (`str`) – Tag of the task; can be used to use the same task multiple times in a single pipeline (every instance of the task needs to have a different tag then to ensure unique task names)
- **reload** (`bool`) – Set True if the Task should be run regardless of whether cached results already exist
- ****kwargs** – Additional task-specific parameters

class Status(*value*)
Encodes current Status of a *Task*

class TaskResult(*status*, *output*, *exception=None*, *traceback=''*)
Result of a task

Parameters

- **status** (*Status*) – Status code
- **output** (`Any`) – Data returned by the task
- **exception** (`Optional[Exception]`) – Exception if one was raised
- **traceback** (`str`) – Traceback of an exception, if one was raised

exception: Optional[Exception] = None

assert_params_complete()
Asserts that the parameter provided in the constructor match those required by the run function.

This is used to raise TypeError already at task creation time (i.e. early and in main thread), not during task execution time.

Returns: None

Return type `None`

description()
Return a complete description of the task.

The description contains the following properties of the task:

- name
- module
- tag
- hash (see *hash()*)
- parameter dictionary

Returns: Dictionary with information describing the task's configuration

Return type `Dict[str, Union[str, Dict, List]]`

hash()
Generate the task's hash

The hash is constructed from the hash of the specific node and the hashes of *all* parent (predecessor) nodes. It is therefore dependent on the pipeline the task is part of. This is used to precisely identify a task in a given context (e.g. when saving and loading pipelines using the *pyperunner.Pipeline.to_file()*

and *pyperunner.Pipeline.from_file()* methods to store/load a pipeline from file). It ensures reproducibility of single pipeline runs.

Returns: Task hash

Return type str

load_output()
Get the output generated by this task

Returns: The output generated by this task

Return type Any

output_exists()
Returns whether an output of this task already exists (i.e. was cached from a previous run).

Returns: If the output of this task already exists

Return type bool

output_filename(*filename='result.dump.gz'*)
Get the full path of the file to which the task's output is written.

The output path is build from the data path (set by the pipeline runner), the task's name and it's hash and is therefore specific to a certain configuration of the task (via it's *hash()*).

Parameters **filename** (str) – Filename of the file to which the task's output is written

Returns: Full path of the file to which the task's output is written

Return type str

abstract run(***kwargs*)
Worker method that must be implemented when subclassing.

This method contains the Tasks execution logic. Note: The

Parameters ****kwargs** – Task-specific parameters

Returns: The result of the task that is passed onto the successor task(s) (e.g. a pandas DataFrame or anything else)

Return type Any

set_data_path(*path*)
Set the task's data path.

This is done by the *Runner* when starting the task.

Parameters **path** (str) – Data path

Returns: None

Return type None

set_output(*output*)
Assigns the output generated by the task.

Parameters **output** (Any) – Output generated by the task (may be load from disk if it's cached).

Returns: None

Return type None

set_reload(*reload*)
Sets the reload parameter that specifies whether this task is forced to run even if its output already exists.

Parameters **reload** (bool) – If the task should forcibly run even if its output already exists

Returns: None

Return type None

set_status(*status*)
Set the task's status (see *Status*)

Parameters **status** (*Status*) – Task status

Returns: None

Return type None

should_run()
Determine if the task should run based on the availability of cached data.

If the task was run previously with the same configuration (see *description()*), the output was stored and can be reloaded. In that case, the task does not need to run and this function returns False. The task can be forced to run by setting the *reload* parameter to True when instantiating the task (see *Task()*).

Returns: If the task should be run by the runner

Return type bool

store_output(*output*)
Store the output of this task.

Parameters **output** (Any) – Output of this task

Returns: None

Return type None

store_params()
Store the parameters of this task to a params.yaml file

Returns: None

Return type None

store_result(*result*)
Store an intermediate result

Parameters **result** (Result) – result to store

Returns: None

Return type None

class pyperunner.**TaskResult**(*name*, *data_path*, *conf*)
Access to the output of a single task

Parameters

- **name** (str) – Task name
- **data_path** (str) – Path where the results are stored
- **conf** (Dict) – Configuration as saved by the task during running

9.3 Pipeline

class pyperunner.**Pipeline**(*name*, *tasks=None*)

add(*task*)
Add a single task as a *primary* task of the pipeline ("root task"), i.e. a task at which the pipeline starts.

Parameters **task** (*Task*) – Task to set as primary (root) task

Returns: None

Return type None

static from_dict(*pipeline_dict*, *compare_hashes=True*)
Create a pipeline from a dictionary (usually saved parameter.yaml file from previous pipeline run)

Parameters

- **pipeline_dict** (Dict) – Dictionary from saved pipeline run (parameter.yaml)
- **compare_hashes** (bool) – Set True if the hashes stored in the pipeline_dict dictionary should be compared with the hashes of the task objects created by this function. Used to ensure that Tasks are created in accordance with the saved pipeline run.

Returns: Pipeline with all tasks as defined by dictionary

Return type *Pipeline*

static from_file(*filename*, *compare_hashes=True*)
Create a pipeline from a stored file yaml file.

Parameters

- **filename** (str) – Filename of the yaml file where the pipeline is stored
- **compare_hashes** (bool) – Set True if the hashes stored in the pipeline_dict dictionary should be compared with the hashes of the task objects created by this function. Used to ensure that Tasks are created in accordance with the saved pipeline run.

Returns: Pipeline with all tasks as defined by the file

Return type *Pipeline*

set_results(*results*)
Stores results of a pipeline run

Parameters **results** (*PipelineResult*) – The pipeline results

Returns: None

Return type None

set_tasks(*tasks*)
Sets the supplied tasks as *primary* tasks of the pipeline ("root tasks"), i.e. tasks at which the pipeline will start (there will be no connection between these tasks)

Parameters **tasks** (List[*Task*]) – Tasks to be set as primary tasks (root tasks)

Returns: None

Return type None

to_dict()
Get a dictionary representation of the directed acyclic graph underlying the pipeline.

Returns: Dictionary representation of the directed acyclic graph underlying the pipeline.

Return type Dict

to_file(*filename*)
Save a representation of the pipeline to file (yaml format)

Parameters **filename** (str) – Filename to store the representation to

Returns: None

Return type None

class pyperunner.**Sequential**(*name*, *tasks=None*)
A purely sequential pipeline with no bifurcations (linear pipeline).

add(*task*)
Add a task to the current end of the pipeline.

Note: The add() method of the Sequential pipeline behaves differently from the *Pipeline.add()* method.

Parameters **task** (*Task*) – Task to add as a successor of the current end of the pipeline.

Returns: None

Return type None

set_tasks(*tasks*)
Set the tasks of the current pipeline.

Note: The tasks are connected in the same order as they are in the supplied list, so make sure these are ordered correctly. Note: The set_tasks() method of the Sequential pipeline behaves differently from the *Pipeline.set_tasks()* method.

Parameters **tasks** (List[*Task*]) – Ordered list of tasks, with first element (tasks[0]) being the first task to run (root task)

Returns: None

Return type None

9.4 Runner

class pyperunner.**Runner**(*data_path*, *log_path*, *process_limit=None*)
Pipeline runner with multiprocessing.

This runner executes all tasks of a given pipeline in order. Each task is started as a separate process, forked from the main process (via *multiprocessing.Process*, see Process). If possible, tasks are run in parallel. This is generally possible for tasks that do not depend on each other in any way (i.e. are not predecessors or successors of one another). The maximum number of concurrrent processes is limited via the process_limit` parameter.

Parameters

- **data_path** (str) – Path where data will be stored
- **log_path** (str) – Path where log files will be stored
- **process_limit** (Optional[int]) – Maximum number of concurrent worker processes

analyze_pipeline(*force_reload=False*)
Marks all nodes (tasks) in the pipeline that need to be run.

A node is required to run if:

- force_reload is True
- the node itself is marked to be run (i.e., the node was created using reload=True in *Task()* or there is no cached result for the node)
- any predecessor node is required to run

Parameters **force_reload** (bool) – If true, all nodes are marked to be run

Returns: None

Return type None

assert_valid_pipeline()

Asserts that the pipeline attribute of this object is set to a valid pipeline object.

Returns: None

Return type None

dequeue()

Get a runnable Task from the queue.

Loops through the task queue and evaluates for each task - if any of the predecessor tasks failed - if all of the predecessor tasks have finished

In the former cases (failure of predecessors), the task is removed from the task list and added to the failed task list itself (required to propagate the error). In the latter case (all predecessors finished), the task is ready to run and as such removed from the queue and returned by the function. Otherwise the function returns None.

Returns: Runnable task, if any, else None

Return type Optional[*Task*]

execution_plan_summary(*print_fn=None*)

Print execution summary of the underlying pipeline DAG in ASCII format.

Note that this may easily take a lot of vertical space if multiple tasks are run in parallel and may not be readable anymore.

Parameters **print_fn** (Optional[Callable]) – Function that prints the execution summary (default: print)

Returns: None

Return type None

finish_tasks()

Clean up after tasks have finished.

Loops through processes that have started but not been removed from the running processes list yet. Next, the following steps are performed

- **If process is dead, it is removed from the list of running tasks**
  - If the processed exited with an error exit code (e.g. out of memory error), that exception is logged
- **Data from the result queue is tried to be read (independent of whether the process is dead or alive)**
  - If data could be read from the queue, the *Runner.process_task_result()* function is called
  - **If no data could be read**

* if the process is still alive, nothing happens
* if the process is dead, an exception is raised

Returns: None

Return type None

generate_run_name()

Get a unique name for the current run of the pipeline.

Used to create the run-specific log path

Returns: pipeline name + current timestamp e.g. "<my-pipeline_210119T222719"

Return type str

get_predecessor_outputs(*task*)

Get the results of all predecessor (parent) task of a task.

Used to feed the outputs (return values) of the parent task to a child class when running it.

Parameters **task** (*Task*) – Task for which the predecessor's outputs are returned.

Returns: List of all parent task outputs

Return type List[Any]

log_exception(*result*)

Provide exception info from a task result to the logger.

Called when processing task results (*Runner.process_task_result()*)

Parameters **result** (*TaskResult*) – Task result

Returns: None

Return type None

pipeline_params_filename()

Get filename of pipeline parameter yaml file.

See *Runner.save_pipeline_params()*

Returns: Filename of pipeline parameter yaml file

Return type str

process_task_result(*task*, *result*)

Process the result object of a task.

When a task has finished running or does not need to be run at all, this function performs the required internal status maintenance:

- Set the task result and status to those of the result object
- Add the task to the finished or the failed tasks and log exceptions, if necessary

Parameters

- **task** (*Task*) – Task of which the result should be processed
- **result** (*TaskResult*) – Result of the given task

Returns: None

Return type None

queue_tasks(*force_reload=False*)
Fills lists of tasks that need to be executed.

Note that there are two lists:

- *tasks_queue* A list of all tasks in order (topological sorting of the underlying DAG)
- *tasks_execute* A list of tasks that need to be executed

Tasks do not need to be executed when their result is cached and the result is not required in a direct child task.

Parameters **force_reload** (bool) – If true, all nodes are marked to be run

Returns: None

Return type None

results()
Get the *PipelineResult* object for the current run.

Returns: PipelineResult object for current run

Return type *PipelineResult*

run(*pipeline=None*, *force_reload=False*, *show_execution_plan=False*)
Run a pipeline.

Main function of the runner. Evaluates which tasks to run and executes them in order. Each task is run in a separate process. Results from each task are collected using multiprocessing.Queue.

Call *Runner.results()* after finishing the run to access the results or alternatively create a *pyperunner.PipelineResult* object explicitly from the parameter.yaml file of the run.

Parameters

- **pipeline** (Optional[*Pipeline*]) – pipeline to be run (or None, if already set via *Runner.set_pipeline()*)
- **force_reload** (bool) – Set true if all tasks should run even if they would not need to
- **show_execution_plan** (bool) – Set true if an execution summary of the underlying pipeline DAG should be displayed.

Returns: None

Return type None

save_pipeline_params()
Stores a complete description of the current pipeline run to a yaml file.

Parameter file contains - run specific information (paths, environment) (see py:func:*pyperunner.environment.get_environment_info*) - name, parameter and parents of each task

Returns: None

Return type None

set_pipeline(*pipeline*)
Sets the pipeline to be run by the runner.

Parameters **pipeline** (*Pipeline*) – Pipeline to be run

Returns: None

Return type None

skip_task(*task*)
Skips execution of a task.

Used for tasks that are neither required to run nor required to load data from (they are completely bypassed). E.g. in a pipeline of tasks a -> b -> c, if only c needs to be run (because it wasn't run so far or because it is forced to run setting reload=True in *Task()*

Parameters **task** (*Task*) – Task to skip

Returns: None

Return type None

start_task(*task*)
Run a task.

Initiates the Process object for the task class and starts it by forking (see Process.run()). Process

Parameters **task** (*Task*) – Task to start

Returns: Process object of the started task

Return type Process

tasks_execute: List[*pyperunner.task.Task*]
A list of tasks that need to be executed

tasks_queue: List[*pyperunner.task.Task*]
A list of all tasks in order (topological sorting of the underlying DAG)

static validate_pipeline(*pipeline*)
Asserts a pipeline conforms to requirements.

Requirements - Every node in the pipeline must have a unique name - The pipeline must be acyclic

Raises exceptions if one of the requirements isn't met by the supplied pipeline.

Parameters **pipeline** (*Pipeline*) – Pipeline to check

Returns: None

Return type None

write_status_image(*fname='status.png'*)
Write an image of the pipeline's DAG to disk.

Creates a PNG image of each of the node and the connections of the DAG associated with the pipeline and writes that to disk

Parameters **fname** (str) – Filename to write to (PNG)

Return type None

9.5 PipelineResult

class pyperunner.**PipelineResult**(*conf*, *data_path=''*)
Accessor of task results of a pipeline run.

Instantiate using the pipeline run configuration or using the from_file method pointing to the parameter.yaml file saved during the pipeline run. Then iterate over the single tasks results or access single task results using array indexing:

```
results = PipelineResult.from_file(fname)

# loop through individual task results
for task_name in pipeline.results():
    print(pipeline[task_name])
```

Parameters

- **conf** (Dict) – Configuration (pipeline.yaml saved in the pipeline run log dir)
- **data_path** (str) – Path where the results are stored

static from_file(*filename*)

Create a PipelineResult object from the parameter.yaml file saved in the pipeline run log path

Parameters **filename** (str) – parameter.yaml file saved in the pipeline run log path

Returns: PipelineResult object

Return type *PipelineResult*

set_data_path(*data_path*)

Sets the data path of the task

Parameters **data_path** (str) – Path where the results are stored

Returns: None

Return type None

task_result(*task*)

Return the results of a certain task.

Parameters **task** (str) – Task name

Returns: Results of the task

Return type Any

CHAPTER

TEN

AUTHORS

Main developer: Gregor Lichtner @glichtner

CHAPTER

ELEVEN

CHANGE LOG

11.1 Version History

0.1.1

- Added LICENSE to source dist

0.1.1

- Fixed setup.py for conda installation

0.1.0

- PypeRunner's first release

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

add() (*pyperunner.Pipeline method*), 29
add() (*pyperunner.Sequential method*), 30
analyze_pipeline() (*pyperunner.Runner method*), 30
assert_params_complete() (*pyperunner.Task method*), 26
assert_valid_pipeline() (*pyperunner.Runner method*), 31

D

dequeue() (*pyperunner.Runner method*), 31
description() (*pyperunner.Task method*), 26

E

exception (*pyperunner.Task.TaskResult attribute*), 26
execution_plan_summary() (*pyperunner.Runner method*), 31

F

finish_tasks() (*pyperunner.Runner method*), 31
from_dict() (*pyperunner.Pipeline static method*), 29
from_file() (*pyperunner.Pipeline static method*), 29
from_file() (*pyperunner.PipelineResult static method*), 35

G

generate_run_name() (*pyperunner.Runner method*), 32
get_predecessor_outputs() (*pyperunner.Runner method*), 32

H

hash() (*pyperunner.Task method*), 26

L

load_output() (*pyperunner.Task method*), 27
log_exception() (*pyperunner.Runner method*), 32

O

output_exists() (*pyperunner.Task method*), 27
output_filename() (*pyperunner.Task method*), 27

P

Pipeline (*class in pyperunner*), 29
pipeline_params_filename() (*pyperunner.Runner method*), 32
PipelineResult (*class in pyperunner*), 34
process_task_result() (*pyperunner.Runner method*), 32

Q

queue_tasks() (*pyperunner.Runner method*), 32

R

results() (*pyperunner.Runner method*), 33
run() (*pyperunner.Runner method*), 33
run() (*pyperunner.Task method*), 27
Runner (*class in pyperunner*), 30

S

save_pipeline_params() (*pyperunner.Runner method*), 33
Sequential (*class in pyperunner*), 30
set_data_path() (*pyperunner.PipelineResult method*), 35
set_data_path() (*pyperunner.Task method*), 27
set_output() (*pyperunner.Task method*), 27
set_pipeline() (*pyperunner.Runner method*), 33
set_reload() (*pyperunner.Task method*), 27
set_results() (*pyperunner.Pipeline method*), 29
set_status() (*pyperunner.Task method*), 28
set_tasks() (*pyperunner.Pipeline method*), 29
set_tasks() (*pyperunner.Sequential method*), 30
should_run() (*pyperunner.Task method*), 28
skip_task() (*pyperunner.Runner method*), 33
start_task() (*pyperunner.Runner method*), 34
store_output() (*pyperunner.Task method*), 28
store_params() (*pyperunner.Task method*), 28
store_result() (*pyperunner.Task method*), 28

T

Task (*class in pyperunner*), 25
Task.Status (*class in pyperunner*), 26

Task.TaskResult (*class in pyperunner*), 26
task_result() (*pyperunner.PipelineResult method*), 35
TaskResult (*class in pyperunner*), 28
tasks_execute (*pyperunner.Runner attribute*), 34
tasks_queue (*pyperunner.Runner attribute*), 34
to_dict() (*pyperunner.Pipeline method*), 29
to_file() (*pyperunner.Pipeline method*), 30

V

validate_pipeline() (*pyperunner.Runner static method*), 34

W

write_status_image() (*pyperunner.Runner method*), 34