Track

Release 0.0.1-c92247e2

Jan 10, 2020

User's Guide

CHAPTER 1

Installation

```
pip install -r requirements
python setup.py install
```

CHAPTER 2

Documentation

```
sphinx-build -W --color -c docs/src/ -b html docs/src/ docs/build/html
(cd docs/build/html && python -m http.server 8000 --bind 127.0.0.1)
```

CHAPTER 3

Overview

```
from track import TrackClient

client = TrackClient('file://client_test.json')
client.set_project(name='test_client')

trial = client.new_trial()
trial.log_arguments(batch_size=256)

with trial:
    trial.log_metrics(step=1, epoch_loss=1)
    trial.log_metrics(accuracy=0.98)

client.save()
client.report()
```

3.1 Overview

Track as 3 kind of objects, Project, Trial Group and Trial.

- **Project** is a top level object that holds all of its trials and groups
- **TrialGroup** is a set of trials. They are used to order trials together. trials can belong to multiple groups
- **Trial** is the object holding all the information about a given training session. the trial object is the backbone of track and it is the object you will have to deal with the most often

3.1.1 Overview

```
from track import TrackClient

client = TrackClient('file://client_test.json')
```

(continues on next page)

(continued from previous page)

```
project = client.set_project(name='paper_78997')
group = client.set_group(name='idea_4573')

trial = client.new_trial(name='final_trial_2', description='almost graduating')
trial.log_arguments(batch_size=256, lr=0.01,momentum=0.99)
trial.log_metadata(gpu='V100')

# start the trial explicitly
with trial:

    for e in range(epochs):

        for batch in dataset:

            # trial helper that compute elapsed time inside a block
            with trial.chrono('batch_time'):
                ...
                loss += ...

        trial.log_metrics(step=e, epoch_loss=loss)

    trial.log_metrics(accuracy=0.98)

client.report()
```

You can find the sample of a report below


```
{
  "revision": 1,
  "name": "final_trial_2",
  "description": "almost graduating",
  "version": "a8c3",
  "tags": {
    "workers": 8,
    "hpo": "byopt"
  },
  "parameters": {
    "batch_size": 32,
    "cuda": true,
    "workers": 0,
    "seed": 0,
    "epochs": 2,
    "arch": "convnet",
    "lr": 0.1,
    "momentum": 0.9,
    "opt_level": "O0",
    "break_after": null,
    "data": "mnist",
    "backend": null
  },
  "metadata": {},
  "metrics": {
    "epoch_loss": {
      "0": 2.306920262972514,
      "1": 2.307889754740397
    }
```


(continues on next page)

(continued from previous page)


```
  },
  "chronos": {
    "runtime": 3142.5199086666107,
    "batch_time": {
      "avg": 0.6737696465350126,
      "min": 0.019209623336791992,
      "max": 445.9658739566803,
      "sd": 12.500646799505962,
      "count": 3751,
      "unit": "s"
    }
  },
  "errors": [],
  "status": {
    "value": 302,
    "name": "Completed"
  }
}
```


3.1.2 Log Metrics

User can log metrics with a step or without. *step* is used as key in a dictionary and should be unique

```
trial.log_metrics(step=e, epoch_loss=loss, metric2=value)

trial.log_metrics(cost=val)
```

3.1.3 Time things

You can easily time things with *chrono*. Do not forget if you are measuring GPU compute time should should synchronize to make sure the computation are done before computing the elapsed time.

```
with trial.chrono('long_compute'):
    sleep(100)
```

3.1.4 Save arbitrary data

You can use metadata to save information on a specific trial that might not be reflected by its parameters

```
trial.log_metadata(had_short_hair_when_running_this_trial=False)
```

3.1.5 Experiment Report

Get a quick overview of all the data that was saved up during training

```
trial.report()
```

3.2 Backends

Track was made to support different backends, you can even implement your own!

3.2.1 Local Backend

Track implements a local storage backend for quick and simple experiments

```
client = TrackClient(f'file://report.json')
```

3.2.2 CockroachDB backend

Track implements a backend that can use a running cockroachdb instance as storage.

```
address = '127.0.0.1
port = 8123
client = TrackClient(f'cockroach://{address}:{port}')
```

3.2.3 Socket backend

Track implements a backend that uses sockets to forward request to a remote server

Server

Simple servers that receive request from the client and forwards all request to another backend. The example below forwards all request to the local backend, allowing to have a single process modifying the file.

```
from track.persistence.socketed import start_track_server

address = '127.0.0.1
port = 8123
layer = 'AES'
start_track_server('file:server_test.json', address, port, backend=layer)
```

Client

Start a client that forwards all request to a remote server

```
username = ...
password = ...
address = '127.0.0.1
port = 8123
layer = 'AES'            # supported AES or (None, i.e put nothing)
client = TrackClient(f'socket://{username}:{password}@{address}:{port}?security_layer=
↪{layer}')
```

3.2.4 Bring Your Own Backend

To implement you own you can simply extend *track.persistence.protocol.Protocol*

```
from track.persistence import register
from track.persistence.protocol import Protocol

class MyOwnBackend(Protocol):
    ....


register('byob', MyOwnBackend)
```

You can then use it naturally

```
client = TrackClient('byob://....)
```

3.3 Simple example

3.3.1 Installation and setup

In this tutorial you will run a very simple MNIST example in pytorch using Track. First, install Track, then install `pytorch`, `torchvision` and clone the PyTorch examples repository:

```
$ pip3 install torch torchvision
$ git clone git@github.com:pytorch/examples.git
```

3.3.2 Adapting the code of MNIST example

After cloning pytorch examples repository, cd to mnist folder:

```
$ cd examples/mnist
```

In main, just after parsing the arguments, you can initialize the track client and create a trial. The client specifies how will the data be saved on your computer, different methods are supported. Once the client is initialized, you can create a new trial.

A trial is a set of data retrieved for a set of arguments.

```
$ ....
$ args = parser.parse_args()
$ client = TrackClient('file:mnist_example.json')
$ trial = client.new_trial(arguments=args)
```

Then you can store any kind of data that you think will be useful. In our example we decided to save the error rate on the test set

```
$ def test(args, model, device, test_loader, trial):
$       ...
$     trial.log_metrics(error_rate=1 - (correct / len(test_loader.dataset)))
```

At the end of training file *mnist_example.json* will be generated holding all the data you saved during training.

3.4 track

3.4.1 track package

Subpackages

track.aggregators package

Submodules

track.aggregators.aggregator module

class track.aggregators.aggregator.**Aggregator**
Bases: object

Attributes

val	Return the last observed value

Methods

lazy(aggregator_t, **kwargs)	Lazily instantiate the underlying aggregator

append	
to_json	

append(*self*, *other*)

static lazy(*aggregator_t*, ***kwargs*)
Lazily instantiate the underlying aggregator

to_json(*self*, *short=False*)

val
Return the last observed value

class track.aggregators.aggregator.**RingAggregator**(*n*, *dtype='f'*)
Bases: *track.aggregators.aggregator.Aggregator*

Saves the *n* last elements. Start overriding the elements once *n* elements is reached

Attributes

val	Return the last observed value

Methods

lazy(n, dtype)	Lazily instantiate the underlying aggregator

append	
to_json	

append(*self*, *other*)

static lazy(*n*, *dtype*)
Lazily instantiate the underlying aggregator

to_json(*self*, *short=False*)

val
Return the last observed value

class track.aggregators.aggregator.**StatAggregator**(*skip_obs=10*)
Bases: *track.aggregators.aggregator.Aggregator*

Compute mean, sd, min, max; does not keep the entire history. This is useful if you are worried about memory usage and the values should not vary much. i.e keeping the entire history is not useful.

Attributes

avg

max

min

sd

sum

total

val Return the last observed value

Methods

lazy(skip)	Lazily instantiate the underlying aggregator

append	
from_json	
to_json	

append(*self*, *other*)

avg

static from_json(*data*)

static lazy(*skip*)
Lazily instantiate the underlying aggregator

max

min

sd

sum

to_json(*self*, *short=False*)

total

val
Return the last observed value

class track.aggregators.aggregator.**TimeSeriesAggregator**
Bases: *track.aggregators.aggregator.Aggregator*

Keeps the entire history of the metric

Attributes

val	Return the last observed value

Methods

lazy()	Lazily instantiate the underlying aggregator

append	
to_json	

append(*self*, *other*)

static lazy()
Lazily instantiate the underlying aggregator

to_json(*self*, *short=False*)

val
Return the last observed value

class track.aggregators.aggregator.**ValueAggregator**(*val=None*)
Bases: *track.aggregators.aggregator.Aggregator*

Does not Aggregate only keeps the latest value

Attributes

val	Return the last observed value

Methods

lazy()	Lazily instantiate the underlying aggregator

append	
to_json	

append(*self*, *other*)

static lazy()
Lazily instantiate the underlying aggregator

to_json(*self*, *short=False*)

val
Return the last observed value

Module contents

track.containers package

Submodules

track.containers.ring module

class track.containers.ring.**RingBuffer**(*size*, *dtype*, *default_val=0*)
Bases: object

Methods

append	
last	
to_list	

append(*self*, *item*)

last(*self*)

to_list(*self*)

track.containers.types module

Module contents

track.dashboard package

Submodules

track.dashboard.dasboard module

Module contents

track.distributed package

Submodules

track.distributed.cockroachdb module

class track.distributed.cockroachdb.**CockRoachDB**(*location*, *addrs*, *join=None*, *clean_on_exit=True*)
Bases: object

cockroach db is a highly resilient database that allow us to remove the Master in a traditional distributed setup.

This spawn a cockroach node that will store its data in *location*

Attributes

build

client_flags

node_id

sql

status

webui

Methods

parse	
start	
stop	
wait	

build

client_flags

node_id

parse (*self*, *properties*, *line*)

sql

start (*self*, *wait=True*)

status

stop (*self*)

wait (*self*)

webui

Module contents

track.persistence package

Submodules

track.persistence.cockroach module

track.persistence.cometml module

track.persistence.local module

exception track.persistence.local.**ConcurrentWrite**(*msg*)
Bases: Exception

class track.persistence.local.**FileProtocol**(*uri*, *strict=True*, *eager=True*)
Bases: *track.persistence.protocol.Protocol*

Local File storage to manage experiments

Parameters

uri: str resource to use to store the experiment *file://my_file.json*

strict: bool forces the storage to be correct. if we use the file protocol as an in-memory storage we might get some inconsistencies we can use this flag to ignore them

eager: bool eagerly update the underlying files. This is necessary if multiple processes are reading from the file

Methods

commit(self[, file_name_override])	Forces to persist the change

add_group_trial
add_project_trial
add_trial_tags
fetch_and_update_group
fetch_and_update_trial
fetch_groups
fetch_projects
fetch_trials
get_project
get_trial
get_trial_group
log_trial_arguments
log_trial_chrono_finish
log_trial_chrono_start
log_trial_finish
log_trial_metadata
log_trial_metrics
log_trial_start
new_project
new_trial
new_trial_group
set_group_metadata
set_trial_status

add_group_trial(*self*, **args*, ***kwargs*)

add_project_trial(*self*, **args*, ***kwargs*)

add_trial_tags(*self*, **args*, ***kwargs*)

commit (*self*, *file_name_override=None*, ***kwargs*)
Forces to persist the change

fetch_and_update_group (*self*, **args*, ***kwargs*)

fetch_and_update_trial (*self*, **args*, ***kwargs*)

fetch_groups (*self*, **args*, ***kwargs*)

fetch_projects (*self*, **args*, ***kwargs*)

fetch_trials (*self*, **args*, ***kwargs*)

get_project (*self*, **args*, ***kwargs*)

get_trial (*self*, **args*, ***kwargs*)

get_trial_group (*self*, **args*, ***kwargs*)

log_trial_arguments (*self*, **args*, ***kwargs*)

log_trial_chrono_finish (*self*, **args*, ***kwargs*)

log_trial_chrono_start (*self*, **args*, ***kwargs*)

log_trial_finish (*self*, **args*, ***kwargs*)

log_trial_metadata (*self*, **args*, ***kwargs*)

log_trial_metrics (*self*, **args*, ***kwargs*)

log_trial_start (*self*, **args*, ***kwargs*)

new_project (*self*, **args*, ***kwargs*)

new_trial (*self*, **args*, ***kwargs*)

new_trial_group (*self*, **args*, ***kwargs*)

set_group_metadata (*self*, **args*, ***kwargs*)

set_trial_status (*self*, **args*, ***kwargs*)

class track.persistence.local.**LockFileRemover** (*filename*)
Bases: *track.utils.signal.SignalHandler*

Methods

atexit	
remove	
sigint	
sigterm	

atexit (*self*)

remove (*self*)

sigint (*self*, *signum*, *frame*)

sigterm (*self*, *signum*, *frame*)

class track.persistence.local.**MultiLock** (*obj*)
Bases: object

track.persistence.local.**execute_query**(*obj*, *query*)
Check if the object *obj* matches the query.

The query is a dictionary specifying constraint on each of the object attributes

track.persistence.local.**lock_atomic_write**(*fun*)

track.persistence.local.**lock_guard**(*readonly*, *atomic=False*)
Protect a function call with a lock. reload the database before the action and save it afterwards

track.persistence.local.**lock_read**(*fun*)

track.persistence.local.**lock_write**(*fun*)

track.persistence.local.**make_lock**(*name*, *eager*)

track.persistence.local.**query_gt**(*obj*, *attrs*, *val*)

track.persistence.local.**query_in**(*obj*, *attrs*, *choices*)

track.persistence.local.**query_lte**(*obj*, *attrs*, *val*)

track.persistence.local.**query_ne**(*obj*, *attrs*, *val*)

track.persistence.multiplexer module

class track.persistence.multiplexer.**ProtocolMultiplexer**(**backends*)
Bases: object

Methods

get_project(self, *args, **kwargs)

add_group_trial	
add_project_trial	
add_trial_tags	
commit	
fetch_and_update_group	
fetch_and_update_trial	
fetch_groups	
fetch_projects	
fetch_trials	
get_trial	
get_trial_group	
log_trial_arguments	
log_trial_chrono_finish	
log_trial_chrono_start	
log_trial_finish	
log_trial_metadata	
log_trial_metrics	
log_trial_start	
new_project	
new_trial	
new_trial_group	
set_trial_status	

add_group_trial(*self*, **args*, ***kwargs*)

add_project_trial(*self*, **args*, ***kwargs*)

add_trial_tags(*self*, **args*, ***kwargs*)

commit(*self*, **args*, ***kwargs*)

fetch_and_update_group(*self*, **args*, ***kwargs*)

fetch_and_update_trial(*self*, **args*, ***kwargs*)

fetch_groups(*self*, **args*, ***kwargs*)

fetch_projects(*self*, **args*, ***kwargs*)

fetch_trials(*self*, **args*, ***kwargs*)

get_project(*self*, **args*, ***kwargs*)

get_trial(*self*, **args*, ***kwargs*)

get_trial_group(*self*, **args*, ***kwargs*)

log_trial_arguments(*self*, **args*, ***kwargs*)

log_trial_chrono_finish(*self*, **args*, ***kwargs*)

log_trial_chrono_start(*self*, **args*, ***kwargs*)

log_trial_finish(*self*, **args*, ***kwargs*)

log_trial_metadata(*self*, **args*, ***kwargs*)

log_trial_metrics(*self*, **args*, ***kwargs*)

log_trial_start(*self*, **args*, ***kwargs*)

new_project(*self*, **args*, ***kwargs*)

new_trial(*self*, **args*, ***kwargs*)

new_trial_group(*self*, **args*, ***kwargs*)

set_trial_status(*self*, **args*, ***kwargs*)

track.persistence.protocol module

class track.persistence.protocol.**Protocol**
Bases: object

Methods

add_group_trial(self, group, trial)	Add a trial to a group
add_project_trial(self, project, trial)	Add a trial to a project
add_trial_tags(self, trial, **kwargs)	Add tags to a trial
commit(self, **kwargs)	Forces to persist the change
fetch_and_update_group(self, query, attr, ...)	Fetch and update a single group
fetch_and_update_trial(self, query, attr, ...)	Fetch and update a single trial
fetch_groups(self, query)	Fetch groups according to a given query
fetch_projects(self, query)	Fetch projects according to a given query
fetch_trials(self, query)	Fetch trials according to a given query
get_project(self, project)	Fetch a project according to the given definition
get_trial(self, trial)	Fetch trials according to a given definition
get_trial_group(self, group)	Fetch a group according to a given definition
log_trial_arguments(self, trial, **kwargs)	Save the arguments a trail
log_trial_chrono_finish(self, trial, name, ...)	Send the end signal for an event
log_trial_chrono_start(self, trial, name, ...)	Send the start signal for an event
log_trial_finish(self, trial, exc_type, ...)	Send the trial end signal
log_trial_metadata(self, trial, aggregator, ...)	Save metadata for a given trials
log_trial_metrics(self, trial, step, ...)	Save metrics for a given trials
log_trial_start(self, trial)	Send the trial start signal
new_project(self, project)	Insert a new project
new_trial(self, trial[, auto_increment])	Insert a new trial
new_trial_group(self, group)	Create a new group
set_trial_status(self, trial, status[, error])	Change trial status

add_group_trial(*self*, *group: track.structure.TrialGroup*, *trial: track.structure.Trial*)
Add a trial to a group

add_project_trial(*self*, *project: track.structure.Project*, *trial: track.structure.Trial*)
Add a trial to a project

add_trial_tags(*self*, *trial*, ***kwargs*)
Add tags to a trial

Parameters

trial: Trial trial reference

kwargs: key value pair of the data to save

commit(*self*, ***kwargs*)

Forces to persist the change

fetch_and_update_group(*self*, *query*, *attr*, **args*, ***kwargs*)
Fetch and update a single group

Parameters

query: Dict dictionary to fetch groups

attr: str name of the update function to call on each selected group

***args:** additional positional arguments for the attr function

****kwargs:** addtional keyword arguments for the attr function

Returns

returns the modified group

fetch_and_update_trial(*self*, *query*, *attr*, **args*, ***kwargs*)
Fetch and update a single trial

Parameters

query: Dict dictionary to fetch trials

attr: str name of the update function to call on each selected trials

***args:** additional positional arguments for the attr function

****kwargs:** addtional keyword arguments for the attr function

Returns

returns the modified trial

fetch_groups(*self*, *query*)
Fetch groups according to a given query

fetch_projects(*self*, *query*)
Fetch projects according to a given query

fetch_trials(*self*, *query*) → List[track.structure.Trial]
Fetch trials according to a given query

get_project(*self*, *project: track.structure.Project*) → Union[track.structure.Project, NoneType]
Fetch a project according to the given definition

Parameters

project: Project project definition used for the lookup

Returns

returns a project object or None

get_trial(*self*, *trial: track.structure.Trial*) → List[track.structure.Trial]
Fetch trials according to a given definition

Parameters

trial: Trial trial definition used for the lookup

get_trial_group(*self*, *group: track.structure.TrialGroup*) → Union[track.structure.TrialGroup, NoneType]
Fetch a group according to a given definition

Parameters

group: TrialGroup group definition used for the lookup

Returns

returns a grouo

log_trial_arguments(*self*, *trial: track.structure.Trial*, ***kwargs*)
Save the arguments a trail

Parameters

trial: Trial trial for which the arguments are for

kwargs: key value pair of arguments

log_trial_chrono_finish(*self*, *trial*, *name*, *exc_type*, *exc_val*, *exc_tb*)
Send the end signal for an event

Parameters

trial: Trial trial sending the event

name: str name of the event

exc_type: Exception object

exec_val Exception value

exc_tb: Traceback

log_trial_chrono_start(*self*, *trial*, *name: str*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = <function StatAggregator.lazy.<locals>.<lambda> at 0x7ff802abcf28>*, *start_callback=None*, *end_callback=None*)
Send the start signal for an event

Parameters

trial: Trial trial sending the event

name: str name of the event

aggregator: Aggregator container used to accumulate elapsed time

start_callback: Callable function called at start time

end_callback: Callable function called at the end

log_trial_finish(*self*, *trial*, *exc_type*, *exc_val*, *exc_tb*)
Send the trial end signal

Parameters

trial: Trial reference to the trial that finished

log_trial_metadata(*self*, *trial: track.structure.Trial*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = <function ValueAggregator.lazy.<locals>.<lambda> at 0x7ff802aabd90>*, ***kwargs*)
Save metadata for a given trials

Parameters

trial: Trial trial reference

kwargs: key value pair of the data to save

log_trial_metrics(*self*, *trial: track.structure.Trial*, *step: <built-in function any> = None*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = None*, ***kwargs*)
Save metrics for a given trials

Parameters

trial: Trial trial reference

kwargs: key value pair of the data to save

log_trial_start(*self*, *trial*)
Send the trial start signal

Parameters

trial: Trial reference to the trial being started

new_project(*self*, *project: track.structure.Project*)
Insert a new project

Parameters

project: Project project definition used for the insert

new_trial(*self*, *trial: track.structure.Trial*, *auto_increment=False*)
Insert a new trial

Parameters

trial: Trial trial definition used for the insert

auto_increment: bool If trial exist increment revision number

Returns

Returns None if Trial already exists and auto_increment is False

new_trial_group(*self*, *group: track.structure.TrialGroup*)
Create a new group

Parameters

group: TrialGroup group definition used for the insert

set_trial_status(*self*, *trial: track.structure.Trial*, *status*, *error=None*)
Change trial status

Parameters

trial: Trial trial reference

status: new status to update the trial too

error: in case the user is changing to a state representing an error it can also provide an error identification string

track.persistence.socketed module

Implement a Remote Logger. Client forwards all the user's request down to the server that executes them one by one.

exception track.persistence.socketed.**RPCCallFailure**(*message*, *trace=None*)
Bases: Exception

class track.persistence.socketed.**ServerSignalHandler**(*server*)
Bases: *track.utils.signal.SignalHandler*

Methods

atexit	
sigint	
sigterm	

sigint(*self*, *signum*, *frame*)

sigterm(*self*, *signum*, *frame*)

class track.persistence.socketed.**SocketClient**(*uri*)
Bases: *track.persistence.protocol.Protocol*

Forwards all the local track requests to the track server that execute the requests and send back the results

Clients can provide a username and password for authentication

Methods

add_group_trial(self, group, trial)	Add a trial to a group
add_project_trial(self, project, trial)	Add a trial to a project
add_trial_tags(self, trial, **kwargs)	Add tags to a trial
authenticate(self, uri)	returns the username and password used for authentication purposes you can override this function to implement a custom authentication method
commit(self, **kwargs)	Forces to persist the change
fetch_and_update_group(self, query, attr, ...)	Fetch and update a single group
fetch_and_update_trial(self, query, attr, ...)	Fetch and update a single trial
fetch_groups(self, query)	Fetch groups according to a given query
fetch_projects(self, query)	Fetch projects according to a given query
fetch_trials(self, query)	Fetch trials according to a given query
get_project(self, project)	Fetch a project according to the given definition
get_trial(self, trial)	Fetch trials according to a given definition
get_trial_group(self, group)	Fetch a group according to a given definition
log_trial_arguments(self, trial, **kwargs)	Save the arguments a trail
log_trial_chrono_finish(self, trial, name, ...)	Send the end signal for an event
log_trial_chrono_start(self, trial, name, ...)	Send the start signal for an event
log_trial_finish(self, trial, exc_type, ...)	Send the trial end signal
log_trial_metadata(self, trial, aggregator, ...)	Save metadata for a given trials
log_trial_metrics(self, trial, step, ...)	Save metrics for a given trials
log_trial_start(self, trial)	Send the trial start signal
new_project(self, project)	Insert a new project
new_trial(self, trial)	Insert a new trial
new_trial_group(self, group)	Create a new group
set_trial_status(self, trial, status[, error])	Change trial status

add_group_trial(*self*, *group: track.structure.TrialGroup*, *trial: track.structure.Trial*)
Add a trial to a group

add_project_trial(*self*, *project: track.structure.Project*, *trial: track.structure.Trial*)
Add a trial to a project

add_trial_tags(*self*, *trial*, ***kwargs*)
Add tags to a trial

Parameters

trial: Trial trial reference

kwargs: key value pair of the data to save

authenticate(*self*, *uri*)
returns the username and password used for authentication purposes you can override this function to implement a custom authentication method

commit(*self*, ***kwargs*)
Forces to persist the change

get_project(*self*, *project: track.structure.Project*)
Fetch a project according to the given definition

Parameters

project: Project project definition used for the lookup

Returns

returns a project object or None

get_trial(*self*, *trial: track.structure.Trial*)
Fetch trials according to a given definition

Parameters

trial: Trial trial definition used for the lookup

get_trial_group(*self*, *group: track.structure.TrialGroup*)
Fetch a group according to a given definition

Parameters

group: TrialGroup group definition used for the lookup

Returns

returns a grouo

log_trial_arguments(*self*, *trial: track.structure.Trial*, ***kwargs*)
Save the arguments a trail

Parameters

trial: Trial trial for which the arguments are for

kwargs: key value pair of arguments

log_trial_chrono_finish(*self*, *trial*, *name*, *exc_type*, *exc_val*, *exc_tb*)
Send the end signal for an event

Parameters

trial: Trial trial sending the event

name: str name of the event

exc_type: Exception object

exec_val Exception value

exc_tb: Traceback

log_trial_chrono_start(*self*, *trial*, *name: str*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = <function StatAggregator.lazy.<locals>.<lambda> at 0x7ff800c1a7b8>*, *start_callback=None*, *end_callback=None*)

Send the start signal for an event

Parameters

trial: Trial trial sending the event

name: str name of the event

aggregator: Aggregator container used to accumulate elapsed time

start_callback: Callable function called at start time

end_callback: Callable function called at the end

log_trial_finish(*self*, *trial*, *exc_type*, *exc_val*, *exc_tb*)

Send the trial end signal

Parameters

trial: Trial reference to the trial that finished

log_trial_metadata(*self*, *trial: track.structure.Trial*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = None*, ***kwargs*)

Save metadata for a given trials

Parameters

trial: Trial trial reference

kwargs: key value pair of the data to save

log_trial_metrics(*self*, *trial: track.structure.Trial*, *step: <built-in function any> = None*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = None*, ***kwargs*)

Save metrics for a given trials

Parameters

trial: Trial trial reference

kwargs: key value pair of the data to save

log_trial_start(*self*, *trial*)

Send the trial start signal

Parameters

trial: Trial reference to the trial being started

new_project(*self*, *project: track.structure.Project*)

Insert a new project

Parameters

project: Project project definition used for the insert

new_trial(*self*, *trial: track.structure.Trial*)

Insert a new trial

Parameters

trial: Trial trial definition used for the insert

auto_increment: bool If trial exist increment revision number

Returns

Returns None if Trial already exists and auto_increment is False

new_trial_group(*self*, *group: track.structure.TrialGroup*)
Create a new group

Parameters

group: TrialGroup group definition used for the insert

set_trial_status(*self*, *trial: track.structure.Trial*, *status*, *error=None*)
Change trial status

Parameters

trial: Trial trial reference

status: new status to update the trial too

error: in case the user is changing to a state representing an error it can also provide an error identification string

class track.persistence.socketed.**SocketServer**(*uri*)
Bases: *track.persistence.protocol.Protocol*

Start a track server inside a asyncio loop

Parameters

uri: str socket://{hostname}:{port}?security_layer={}&backend={protocol} with

Users inherit this class to implement their own custom authentication

Methods

add_group_trial(self, group, trial)	Add a trial to a group
add_project_trial(self, project, trial)	Add a trial to a project
add_trial_tags(self, trial, **kwargs)	Add tags to a trial
authenticate(self, reader, username, password)	User defined authentication function
commit(self, **kwargs)	Forces to persist the change
fetch_and_update_group(self, query, attr, ...)	Fetch and update a single group
fetch_and_update_trial(self, query, attr, ...)	Fetch and update a single trial
fetch_groups(self, query)	Fetch groups according to a given query
fetch_projects(self, query)	Fetch projects according to a given query
fetch_trials(self, query)	Fetch trials according to a given query
get_project(self, project)	Fetch a project according to the given definition
get_trial(self, trial)	Fetch trials according to a given definition
get_trial_group(self, group)	Fetch a group according to a given definition
log_trial_arguments(self, trial, **kwargs)	Save the arguments a trail
log_trial_chrono_finish(self, trial, name, ...)	Send the end signal for an event
log_trial_chrono_start(self, trial, name, ...)	Send the start signal for an event

Continued on next page

Table 10 – continued from previous page

log_trial_finish(self, trial, exc_type, ...)	Send the trial end signal
log_trial_metadata(self, trial, aggregator, ...)	Save metadata for a given trials
log_trial_metrics(self, trial, step, ...)	Save metrics for a given trials
log_trial_start(self, trial)	Send the trial start signal
new_project(self, project)	Insert a new project
new_trial(self, trial[, auto_increment])	Insert a new trial
new_trial_group(self, group)	Create a new group
process_args(self, args[, cache])	replace ids by their object reference so the backend modifies the objects and not a copy
run_server(self)	
set_trial_status(self, trial, status[, error])	Change trial status

close	
close_connection	
exec	
get_username	
handle_client	
is_authenticated	
wait_closed	

authenticate(*self*, *reader*, *username*, *password*)
User defined authentication function

Parameters

reader: StreamReader client socket / reader, can be used to link client socket -> username

username: str client username

password: str client password

close(*self*)

static close_connection(*writer*)

commit(*self*, ***kwargs*)
Forces to persist the change

exec(*self*, *reader*, *writer*, *proc_name*, *proc*, *args*, *cache=None*)

get_username(*self*, *reader*)

handle_client(*self*, *reader*, *writer*)

is_authenticated(*self*, *reader*)

process_args(*self*, *args*, *cache=None*)
replace ids by their object reference so the backend modifies the objects and not a copy

run_server(*self*)

static wait_closed(*writer*)

track.persistence.socketed.**read**(*reader*, *timeout=None*)

track.persistence.socketed.**recv**(*socket*, *timeout=None*)

track.persistence.socketed.**send**(*socket*, *msg*)

track.persistence.socketed.**start_track_server**(*protocol*, *hostname*, *port*, *security_layer=None*)

Start a track server inside a asyncio loop

Parameters

protocol: str URI that defines which backend to forward the request to

hostname: str server host name

port: int server port to listen to

security_layer: str backend used for encryption (only AES is supported)

track.persistence.socketed.**to_bytes**(*message*) → bytes

track.persistence.socketed.**to_obj**(*message: bytes*) → <built-in function any>

track.persistence.socketed.**write**(*writer*, *msg*)

track.persistence.storage module

class track.persistence.storage.**LocalStorage**(*target_file: str = None*, *_objects: Dict[uuid.UUID, <built-in function any>] = <factory>*, *_projects: Set[uuid.UUID] = <factory>*, *_groups: Set[uuid.UUID] = <factory>*, *_trials: Set[uuid.UUID] = <factory>*, *_project_names: Dict[str, uuid.UUID] = <factory>*, *_group_names: Dict[str, uuid.UUID] = <factory>*, *_trial_names: Dict[str, uuid.UUID] = <factory>*, *_old_rev_tags: Dict[str, int] = <factory>*)

Bases: object

Attributes

group_names

groups

objects

project_names

projects

target_file

trials

Methods

reload(self[, filename])	Reload storage and discard current objects
smart_reload(self[, filename])	Updates current objects with new data

commit	
get_current_version_tag	
get_previous_version_tag	

commit (*self*, *file_name_override=None*, ***kwargs*)

get_current_version_tag (*self*, *obj*)

get_previous_version_tag (*self*, *obj*)

group_names

groups

objects

project_names

projects

reload (*self*, *filename=None*)
Reload storage and discard current objects

smart_reload (*self*, *filename=None*)
Updates current objects with new data

target_file = None

trials

track.persistence.storage.**load_database** (*json_name*)

track.persistence.utils module

track.persistence.utils.**parse_options** (*options*)

track.persistence.utils.**parse_uri** (*uri*)
Parse a URI and returns a dictionary from it

`scheme://[//authority]path[?query][#fragment]` with `authority = [userinfo@]host[:port]`

Module contents

track.persistence.**get_protocol** (*backend_name*)
proto://arg

track.persistence.**make_cockroach_protocol** (*uri*)

track.persistence.**make_comet_ml** (*uri*)

track.persistence.**make_ephemeral_protocol** (*uri*)

track.persistence.**make_local** (*uri*, *strict=True*, *eager=True*)

track.persistence.**make_mongodb_protocol** (*uri*)

track.persistence.**make_pickled_protocol** (*uri*)

track.persistence.**make_socket_protocol** (*uri*)

track.persistence.**register** (*name*, *proto*)

track.utils package

Submodules

track.utils.debug module

track.utils.debug.**print_stack**(*msg='–'*)

track.utils.delay module

class track.utils.delay.**DelayedCall**(*fun*, *kwargs*)
Bases: object

Delay a call until later

Methods

__call__(self, *args, **kwargs)	Call self as a function.

add_arguments	
get_future	

add_arguments(*self*, ***kwargs*)

get_future(*self*)

class track.utils.delay.**Future**(*promise*)
Bases: object

Methods

get	
is_ready	

get(*self*)

is_ready(*self*)

exception track.utils.delay.**FutureIsNotReady**
Bases: Exception

track.utils.delay.**delay_call**(*fun*, ***kwargs*)

track.utils.delay.**is_delayed_call**(*obj*)

track.utils.encrypted module

class track.utils.encrypted.**EncryptedSocket**(**args*, ***kwargs*)
Bases: socket.socket

Socket with an encrypted layer

Attributes

family Read-only access to the address family for this socket.

proto the socket protocol

timeout the socket timeout

type Read-only access to the socket type.

Methods

accept(self)	Accept an incoming connection & initialize the encryption layer for that client
bind(address)	Bind the socket to a local address.
close()	Close the socket.
connect(address)	Connect the socket to a remote address.
connect_ex()	This is like connect(address), but returns an error code (the errno value) instead of raising an exception when an error occurs.
detach(self)	Close the socket object without closing the underlying file descriptor.
dup(self)	Duplicate the socket.
fileno()	Return the integer file descriptor of the socket.
get_inheritable(self)	Get the inheritable flag of the socket
getblocking()	Returns True if socket is in blocking mode, or False if it is in non-blocking mode.
getpeername()	Return the address of the remote endpoint.
getsockname()	Return the address of the local endpoint.
getsockopt()	Get a socket option.
gettimeout()	Returns the timeout in seconds (float) associated with socket operations.
listen([backlog])	Enable a server to accept connections.
makefile(self[, mode, buffering, encoding, ...])	The arguments are as for io.open() after the filename, except the only supported mode values are 'r' (default), 'w' and 'b'.
recv(self, buffersize, flags[, context])	Receive up to buffersize bytes from the socket.
recv_into()	A version of recv() that stores its data into a buffer rather than creating a new string.
recvfrom(buffersize[, flags])	Like recv(buffersize, flags) but also return the sender's address info.
recvfrom_into(buffer[, nbytes[, flags]])	Like recv_into(buffer[, nbytes[, flags]]) but also return the sender's address info.
recvmsg(bufsize[, ancbufsize[, flags]])	Receive normal data (up to bufsize bytes) and ancillary data from the socket.
recvmsg_into(buffers[, ancbufsize[, flags]])	Receive normal data and ancillary data from the socket, scattering the non-ancillary data into a series of buffers.
send(self, data, flags)	Send a data string to the socket.
sendall(data[, flags])	Send a data string to the socket.

Continued on next page

Table 13 – continued from previous page

sendfile(self, file[, offset, count])	Send a file until EOF is reached by using high-performance os.sendfile() and return the total number of bytes which were sent.
sendmsg()	Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message.
sendmsg_afalg([msg], *, op[, iv[, assoclen[])	Set operation mode, IV and length of associated data for an AF_ALG operation socket.
sendto()	Like send(data, flags) but allows specifying the destination address.
set_inheritable(self, inheritable)	Set the inheritable flag of the socket
setblocking(flag)	Set the socket to blocking (flag is true) or non-blocking (false).
setsockopt(level, option, value, option, ...)	Set a socket option.
settimeout(timeout)	Set a timeout on socket operations.
shutdown(flag)	Shut down the reading side of the socket (flag == SHUT_RD), the writing side of the socket (flag == SHUT_WR), or both ends (flag == SHUT_RDWR).

readsize

accept (*self*)
Accept an incoming connection & initialize the encryption layer for that client

Returns

returns (socket, addr) of the client

readsize (*self*)

recv (*self*, *buffersize*, *flags: int = 0*, *context=None*)
Receive up to buffersize bytes from the socket. For the optional flags argument, see the Unix manual. When no data is available, block until at least one byte is available or until the remote end is closed. When the remote end is closed and all data is read, return the empty string.

send (*self*, *data: bytes*, *flags: int = 0*) → int
Send a data string to the socket. For the optional flags argument, see the Unix manual. Return the number of bytes sent; this may be less than len(data) if the network is busy.

sendall (*data*[, *flags*])
Send a data string to the socket. For the optional flags argument, see the Unix manual. This calls send() repeatedly until all data is sent. If an error occurs, it's impossible to tell how much data has been sent.

track.utils.encrypted.**wrap_socket** (*sock*, *server_side=False*, *handshaked=False*)

track.utils.eta module

class track.utils.eta.**EstimatedTime** (*stat_timer: track.utils.stat.StatStream, total: Union[int, List[int]], start: int = 0, name: str = None*)

Bases: object

Compute estimated time to arrival given average time and remaining steps

Examples

```
>>> timer = StatStream()
>>> total = (10, 1000)
>>> eta = EstimatedTime(timer, total)
>>> eta.estimate_time((1, 2))
```

Attributes

total

Methods

count(item[, offset])	Return the current iteration it given the completion of each steps
elapsed(self, unit)	Return the elapsed time since the class was created
estimated_time(self, step, unit)	Estimate the time remaining before the end of the computation
set_totals(self, t)	Set the total number of iteration for each step
show_eta(self, step[, msg, show])	Print the estimate time until the processing is done

static count (*item*, *offset=0*)
Return the current iteration it given the completion of each steps

elapsed (*self*, *unit: int = 60*)
Return the elapsed time since the class was created

estimated_time (*self*, *step: int*, *unit: int = 60*)
Estimate the time remaining before the end of the computation

set_totals (*self*, *t*)
Set the total number of iteration for each step

show_eta (*self*, *step*, *msg=''*, *show=True*)
Print the estimate time until the processing is done

total

track.utils.eta.**get_time** (*time: track.utils.stat.StatStream*)

track.utils.eta.**to_list** (*item*)

track.utils.log module

track.utils.log.**get_log_record_constructor** ()

track.utils.log.**make_logger** (*name*)

track.utils.log.**set_log_level** (*level=20*)

track.utils.out module

class track.utils.out.**RingOutputDecorator** (*file=None*, *n_entries=50*)
Bases: object

Methods

flush	
out	
output	
raw	
write	

flush (*self*)

out (*self*)

output (*self*)

raw (*self*)

write (*self*, *string*)

track.utils.signal module

class track.utils.signal.**SignalHandler**
Bases: object

Methods

atexit	
sigint	
sigterm	

atexit (*self*)

sigint (*self*, *signum*, *frame*)

sigterm (*self*, *signum*, *frame*)

track.utils.stat module

class track.utils.stat.**StatStream**(*drop_first_obs=10*)
Bases: object

Sharable object

Store the sum of the observations amd the the sum of the observations squared The first few observations are discarded (usually slower than the rest)

The average and the standard deviation is computed at the user's request

In order to make the computation stable we store the first observation and subtract it to every other observations. The idea is if x ~ N(mu, sigma) x - x0 and the sum of x - x0 should be close(r) to 0 allowing for greater precision; without that trick *var* was getting negative on some iteration.

Attributes

avg

count

current_count

current_obs

drop_obs

first_obs

max

min

sd

sum

sum_sqr

total

val

var

Methods

from_dict	
state_dict	
to_array	
to_dict	
to_json	
update	

avg

count

current_count

current_obs

drop_obs

first_obs

static from_dict(*data*)

max

min

sd

state_dict(*self*)

sum

sum_sqr

to_array(*self*, *transform=None*)

to_dict(*self*)

to_json(*self*)

total

update(*self*, *val*, *weight=1*)

val

var

class track.utils.stat.**StatStreamStruct**
Bases: _ctypes.Structure

Attributes

current_count Structure/Union member

current_obs Structure/Union member

drop_obs Structure/Union member

first_obs Structure/Union member

max Structure/Union member

min Structure/Union member

sum Structure/Union member

sum_sqr Structure/Union member

current_count
Structure/Union member

current_obs
Structure/Union member

drop_obs
Structure/Union member

first_obs
Structure/Union member

max
Structure/Union member

min
Structure/Union member

sum
Structure/Union member

sum_sqr
Structure/Union member

track.utils.system module

track.utils.system.**get_gpu_name**()

track.utils.throttle module

class track.utils.throttle.**ThrottleRepeatedCalls**(*fun: Callable[[A], R], every=10*)
Bases: object

Limit how often the function *fun* is called in number of times called

Methods

__call__(self, *args, **kwargs)	Call self as a function.

class track.utils.throttle.**Throttler**(*fun: Callable[[A], R], throttle=1*)
Bases: object

Limit how often the function *fun* is called by calling it only every *throttle* time it has been called

Methods

__call__(self, *args, **kwargs)	Call self as a function.

class track.utils.throttle.**TimeThrottler**(*fun: Callable[[A], R], every=10*)
Bases: object

Limit how often the function *fun* is called in seconds

Methods

__call__(self, *args, **kwargs)	Call self as a function.

track.utils.throttle.**is_throttled**(*fun: Callable[[~A], ~R]*) → bool

track.utils.throttle.**throttle_repeated**(*fun: Callable[[~A], ~R], every=None*) → Callable[[~A], Union[~R, NoneType]]

track.utils.throttle.**throttled**(*fun: Callable[[~A], ~R], throttle=None, every=None*) → Callable[[~A], Union[~R, NoneType]]

Module contents

exception track.utils.**ItemNotFound**
Bases: Exception

track.utils.**listen_socket**(*add*, *port*, *backend=None*)

track.utils.**open_socket**(*add*, *port*, *backend=None*)

Submodules

track.chrono module

class track.chrono.**ChronoContext**(*acc: track.aggregators.aggregator.Aggregator*, *start_callback: Callable = None*, *end_callback: Callable = None*)

Bases: object

Sync is a function that can be set to make the timer wait before ending. This is useful when timing async calls like cuda calls

track.client module

class track.client.**TrackClient**(*backend='none'*)

Bases: object

TrackClient. A client tracks a single Trial being ran

Parameters

backend: str Storage backend to use

Methods

add_tags(self, **kwargs)	Insert tags to current trials
get_arguments(self, args, ...[, show])	See log_arguments() for possible arguments
get_device()	Helper function that returns a cuda device if available else a cpu
log_arguments(self, args, ...[, show])	Store the arguments that was used to run the trial.
new_trial(self[, force])	Create a new trial
report(self[, short])	Print a digest of the logged metrics
save(self[, file_name_override])	Saved logged metrics into a json file
set_group(self, group, NoneType] = None, ...)	Set or create a new group
set_project(self, project, NoneType] = None, ...)	Set or create a new project
set_trial(self, trial, NoneType] = None, ...)	Set a new trial
set_version(self[, version])	Compute the version tag from the function call stack.

finish	
start	

add_tags(*self*, ***kwargs*)

Insert tags to current trials

finish(*self*, *exc_type=None*, *exc_val=None*, *exc_tb=None*)

get_arguments(*self*, *args: Union[argparse.ArgumentParser, argparse.Namespace, Dict] = None*, *show=False*, ***kwargs*) → argparse.Namespace

See log_arguments() for possible arguments

static get_device()

Helper function that returns a cuda device if available else a cpu

log_arguments(*self*, *args: Union[argparse.ArgumentParser, argparse.Namespace, Dict] = None*, *show=False*, ***kwargs*) → argparse.Namespace

Store the arguments that was used to run the trial.

Parameters

args: Union[ArgumentParser, Namespace, Dict] save up the trial's arguments

show: bool print the arguments on the command line

kwargs more trial's arguments

Returns

returns the trial's arguments

new_trial(*self*, *force=False*, ***kwargs*)
Create a new trial

Parameters

force: bool by default once the trial is set it cannot be changed. use force to override this behaviour.

kwargs: See *Trial()* for possible arguments

Returns

returns a trial logger

report(*self*, *short=True*)
Print a digest of the logged metrics

save(*self*, *file_name_override=None*)
Saved logged metrics into a json file

set_group(*self*, *group: Union[track.structure.TrialGroup, NoneType] = None*, *force: bool = False*, *get_only: bool = False*, ***kwargs*)
Set or create a new group

Parameters

group: Optional[TrialGroup] project definition you can use to create or set the project

force: bool by default once the trial group is set it cannot be changed. use force to override this behaviour.

get_only: bool if true does not insert the group if missing. default to false

kwargs arguments used to create a `TrialGroup` object if no TrialGroup object were provided. See *TrialGroup()* for possible arguments

Returns

returns created trial group

set_project(*self*, *project: Union[track.structure.Project, NoneType] = None*, *force: bool = False*, *get_only: bool = False*, ***kwargs*)
Set or create a new project

Parameters

project: Optional[Project] project definition you can use to create or set the project

force: bool by default once the project is set it cannot be changed. use force to override this behaviour.

get_only: bool if true does not insert the project if missing. default to false

kwargs arguments used to create a `Project` object if no project object were provided See *Project()* for possible arguments

Returns

returns created project

set_trial(*self*, *trial: Union[track.structure.Trial, NoneType] = None*, *force: bool = False*, ***kwargs*)
Set a new trial

Parameters

trial: Optional[Trial] project definition you can use to create or set the project

force: bool by default once the trial is set it cannot be changed. use force to override this behaviour.

kwargs: {uid, hash, revision} arguments used to create a `Trial` object if no Trial object were provided. You should specify *uid* or the pair *(hash, revision)*. See *Trial()* for possible arguments

Returns

returns a trial logger

set_version(*self*, *version=None*, *version_fun: Callable[[], str] = None*)
Compute the version tag from the function call stack. Defaults to compute the hash of the executed file

Parameters

version: str version string you want to use for the trial

version_fun: Callable[[], str] version function to call to set the trial version

start(*self*)

exception track.client.**TrialDoesNotExist**
Bases: `Exception`

track.configuration module

track.configuration.**find_configuration**(*file=None*)

track.configuration.**options**(*key*, *default=<track.configuration._DefaultNone object at 0x7ff802a9de80>*)

track.configuration.**reset_configuration**()

track.logger module

class track.logger.**LogSignalHandler**(*logger*)
Bases: *track.utils.signal.SignalHandler*

Methods

atexit	
sigint	
sigterm	

atexit(*self*)

sigint(*self*, *signum*, *frame*)

sigterm(*self*, *signum*, *frame*)

class track.logger.**LoggerChronoContext**(*protocol*, *trial*, *acc=s<{'avg': 0.0, 'min': inf, 'max': -inf, 'sd': 0.0, 'count': 1, 'unit': 's'}>*, *name=None*, ***kwargs*)

Bases: object

class track.logger.**TrialLogger**(*trial: track.structure.Trial*, *protocol: track.persistence.protocol.Protocol*)

Bases: object

Unified logger interface. This object should be created through the *TrackClient* interface

Parameters

trial: Trial the trial that the logger modifies

protocol: Protocol the storage protocol used to persist the log calls

Methods

capture_output(self[, output_size])	capture standard output
chrono(self, name, aggregator, ... [, ...])	Start a timer to measure the time spent in that block
finish(self[, exc_type, exc_val, exc_tb])	finish trial, record end time and set the trial status to completed or interrupted
log_arguments(self, **kwargs)	log the trial arguments.
log_metadata(self, aggregator, ...)	insert metadata value inside a trial
log_metrics(self, step, aggregator, ...)	insert metrics values inside a trial
set_status(self, status[, error])	update trial status
start(self)	Start trial, records start time and set the trial status to running

add_tags	
log_code	
log_directory	
log_file	
set_eta_total	
show_eta	

add_tags(*self*, ***kwargs*)

capture_output(*self*, *output_size=50*)
capture standard output

chrono(*self*, *name: str*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = <function StatAggregator.lazy.<locals>.<lambda> at 0x7ff802ac9ea0>*, *start_callback=None*, *end_callback=None*)
Start a timer to measure the time spent in that block

Parameters

name: str name of the timer

aggregator: how to save the values, by default it uses the StatAggregator and only the mean, sd, max, min values are kept once the training is done

start_callback: Callable function that is called once the timer starts

end_callback: Callable function that is called once the timer ends

Returns

returns a context manager that represents the timer

finish(*self*, *exc_type=None*, *exc_val=None*, *exc_tb=None*)
finish trial, record end time and set the trial status to completed or interrupted

log_arguments(*self*, ***kwargs*)
log the trial arguments. This function has not effect if the trial was already created.

log_code(*self*)

log_directory(*self*, *name*, *recursive=False*)

log_file(*self*, *file_name*)

log_metadata(*self*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = None*, ***kwargs*)
insert metadata value inside a trial

Parameters

kwargs: dictionary of metrics (metadata_name: value)

log_metrics(*self*, *step: <built-in function any> = None*, *aggregator: Callable[[], track.aggregators.aggregator.Aggregator] = None*, ***kwargs*)
insert metrics values inside a trial

Parameters

step: any a value representing a training step (could be epoch, timestamp, . . .)

kwargs: dictionary of metrics (metric_name: value)

aggregator: Optional[Callable[[], Aggregator]] how to store the values locally

set_eta_total(*self*, *t*)

set_status(*self*, *status*, *error=None*)
update trial status

show_eta(*self*, *step: int*, *timer: track.utils.stat.StatStream*, *msg: str = ''*, *throttle=None*, *every=None*, *no_print=False*)

start(*self*)
Start trial, records start time and set the trial status to running

track.serialization module

class track.serialization.**SerializerAspect**
Bases: object

Methods

from_json	
to_json	

from_json(*self*, *obj*)

to_json(*self*, *obj: <built-in function any>*, *short=False*)

class track.serialization.**SerializerChronoContext**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
to_json	

to_json(*self*, *obj: <built-in function any>*, *short=False*)

class track.serialization.**SerializerDatetime**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
to_json	

to_json(*self*, *obj: datetime.datetime*, *short=False*)

class track.serialization.**SerializerProject**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
to_json	

from_json(*self*, *obj*)

to_json(*self*, *obj: track.structure.Project*, *short=False*)

class track.serialization.**SerializerStatStream**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
to_json	

from_json(*self*, *obj*, *short=False*)

class track.serialization.**SerializerStatus**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
to_json	

to_json (*self*, *obj: track.structure.Status*, *short=False*)

class track.serialization.**SerializerTrial**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
to_json	

from_json (*self*, *obj*)

ignore_meta = {'_last_change', '_update_count', 'heartbeat'}

ignore_short = {'uid', 'hash', 'dtype', 'project_id', 'group_id'}

to_json (*self*, *obj: track.structure.Trial*, *short=False*)

class track.serialization.**SerializerTrialGroup**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
maybe_unflatten	
to_json	

from_json (*self*, *obj*)

static maybe_unflatten (*v*)

to_json (*self*, *obj: track.structure.TrialGroup*, *short=False*)

class track.serialization.**SerializerUUID**
Bases: *track.serialization.SerializerAspect*

Methods

from_json	
to_json	

to_json (*self*, *obj: uuid.UUID*, *short=False*)

track.serialization.**from_json** (*obj: Dict[str, <built-in function any>], dtype=None*) → <built-in function any>

track.serialization.**to_json** (*k: <built-in function any>*, *short=False*)

track.structure module

hold basic data type classes that all backends need to implement

class track.structure.**CustomStatus**(*name*, *value*)

Bases: object

Attributes

name

value

name

value

class track.structure.**Project**(*_uid: str = None*, *name: Union[str, NoneType] = None*, *description: Union[str, NoneType] = None*, *metadata: Dict[str, any] = <factory>*, *groups: Set[track.structure.TrialGroup] = <factory>*, *trials: Set[track.structure.Trial] = <factory>*) → None

Bases: object

Set of Trial Groups & trials If projects define tags than all children inherit those tags. children cannot override the tag of a parent

Attributes

description

name

uid

Methods

compute_uid	

compute_uid(*self*) → str

description = None

name = None

uid

class track.structure.**Status**

Bases: enum.Enum

An enumeration.

Broken = 203

Completed = 302

CreatedGroup = 0

ErrorGroup = 200

Exception = 202

FinishedGroup = 300

Interrupted = 201

Running = 101

RunningGroup = 100

Suspended = 301

class track.structure.**Trial**(*_hash: str = None*, *revision: int = 0*, *name: Union[str, NoneType] = None*, *description: Union[str, NoneType] = None*, *tags: Dict[str, any] = <factory>*, *version: Union[str, NoneType] = None*, *group_id: Union[int, NoneType] = None*, *project_id: Union[int, NoneType] = None*, *parameters: Dict[str, any] = <factory>*, *metadata: Dict[str, any] = <factory>*, *metrics: Dict[str, any] = <factory>*, *chronos: Dict[str, any] = <factory>*, *status: Union[track.structure.Status, NoneType] = <Status.CreatedGroup: 0>*, *errors: List[str] = <factory>*) → None

Bases: object

A single training run

Attributes

description

group_id

hash

name

project_id

uid

version

Methods

compute_hash	

compute_hash(*self*) → str

description = None

group_id = None

hash

name = None

project_id = None

revision = 0

status = 0

uid

version = None

class track.structure.**TrialGroup**(*_uid: str = None*, *name: Union[str, NoneType] = None*, *description: Union[str, NoneType] = None*, *metadata: Dict[str, any] = <factory>*, *trials: Set[track.structure.Trial] = <factory>*, *project_id: Union[int, NoneType] = None*) → None

Bases: object

Namespace / Set of trials

Attributes

description

name

project_id

uid

Methods

compute_uid	

compute_uid(*self*) → str

description = None

name = None

project_id = None

uid

track.structure.**get_current_project**()

track.structure.**get_current_trial**()

track.structure.**set_current_project**(*project*)

track.structure.**set_current_trial**(*trial*)

track.structure.**status**(*name=None*, *value=None*)

track.versioning module

track.versioning.**compute_hash**(**args*, ***kwargs*)

track.versioning.**compute_version**(*files: List[str]*) → str

track.versioning.**default_version_hash**()
get the current stack frames and from the file compute the version

track.versioning.**get_file_version**(*file_name: str*) → str
hash the file using sha256, used in combination with get_git_version to version non committed modifications

track.versioning.**get_git_version**(*module*) → Tuple[str, str]

track.versioning.**is_iterable**(*iterable*)

Module contents

class track.**TrackClient**(*backend='none'*)

Bases: object

TrackClient. A client tracks a single Trial being ran

Parameters

backend: str Storage backend to use

Methods

add_tags(self, **kwargs)	Insert tags to current trials
get_arguments(self, args, ...[, show])	See log_arguments() for possible arguments
get_device()	Helper function that returns a cuda device if available else a cpu
log_arguments(self, args, ...[, show])	Store the arguments that was used to run the trial.
new_trial(self[, force])	Create a new trial
report(self[, short])	Print a digest of the logged metrics
save(self[, file_name_override])	Saved logged metrics into a json file
set_group(self, group, NoneType] = None, ...)	Set or create a new group
set_project(self, project, NoneType] = None, ...)	Set or create a new project
set_trial(self, trial, NoneType] = None, ...)	Set a new trial
set_version(self[, version])	Compute the version tag from the function call stack.

finish	
start	

add_tags(*self*, ***kwargs*)

Insert tags to current trials

finish(*self*, *exc_type=None*, *exc_val=None*, *exc_tb=None*)

get_arguments(*self*, *args: Union[argparse.ArgumentParser, argparse.Namespace, Dict] = None*, *show=False*, ***kwargs*) → argparse.Namespace

See log_arguments() for possible arguments

static get_device()

Helper function that returns a cuda device if available else a cpu

log_arguments(*self*, *args: Union[argparse.ArgumentParser, argparse.Namespace, Dict] = None*, *show=False*, ***kwargs*) → argparse.Namespace

Store the arguments that was used to run the trial.

Parameters

args: Union[ArgumentParser, Namespace, Dict] save up the trial's arguments

show: bool print the arguments on the command line

kwargs more trial's arguments

Returns

returns the trial's arguments

new_trial(*self*, *force=False*, ***kwargs*)
Create a new trial

Parameters

force: bool by default once the trial is set it cannot be changed. use force to override this behaviour.

kwargs: See *Trial()* for possible arguments

Returns

returns a trial logger

report(*self*, *short=True*)
Print a digest of the logged metrics

save(*self*, *file_name_override=None*)
Saved logged metrics into a json file

set_group(*self*, *group: Union[track.structure.TrialGroup, NoneType] = None*, *force: bool = False*, *get_only: bool = False*, ***kwargs*)
Set or create a new group

Parameters

group: Optional[TrialGroup] project definition you can use to create or set the project

force: bool by default once the trial group is set it cannot be changed. use force to override this behaviour.

get_only: bool if true does not insert the group if missing. default to false

kwargs arguments used to create a *TrialGroup* object if no TrialGroup object were provided. See *TrialGroup()* for possible arguments

Returns

returns created trial group

set_project(*self*, *project: Union[track.structure.Project, NoneType] = None*, *force: bool = False*, *get_only: bool = False*, ***kwargs*)
Set or create a new project

Parameters

project: Optional[Project] project definition you can use to create or set the project

force: bool by default once the project is set it cannot be changed. use force to override this behaviour.

get_only: bool if true does not insert the project if missing. default to false

kwargs arguments used to create a *Project* object if no project object were provided See *Project()* for possible arguments

Returns

returns created project

set_trial(*self*, *trial: Union[track.structure.Trial, NoneType] = None*, *force: bool = False*, ***kwargs*)
Set a new trial

Parameters

trial: Optional[Trial] project definition you can use to create or set the project

force: bool by default once the trial is set it cannot be changed. use force to override this behaviour.

kwargs: {uid, hash, revision} arguments used to create a *Trial* object if no Trial object were provided. You should specify *uid* or the pair *(hash, revision)*. See `Trial()` for possible arguments

Returns

returns a trial logger

set_version(*self*, *version=None*, *version_fun: Callable[[], str] = None*)
Compute the version tag from the function call stack. Defaults to compute the hash of the executed file

Parameters

version: str version string you want to use for the trial

version_fun: Callable[[], str] version function to call to set the trial version

start(*self*)

class track.**Project**(*_uid: str = None*, *name: Union[str, NoneType] = None*, *description: Union[str, NoneType] = None*, *metadata: Dict[str, any] = <factory>*, *groups: Set[track.structure.TrialGroup] = <factory>*, *trials: Set[track.structure.Trial] = <factory>*) → None
Bases: `object`

Set of Trial Groups & trials If projects define tags than all children inherit those tags. children cannot override the tag of a parent

Attributes

description

name

uid

Methods

compute_uid	

compute_uid(*self*) → str

description = None

name = None

uid

class track.**TrialGroup**(*_uid: str = None*, *name: Union[str, NoneType] = None*, *description: Union[str, NoneType] = None*, *metadata: Dict[str, any] = <factory>*, *trials: Set[track.structure.Trial] = <factory>*, *project_id: Union[int, NoneType] = None*) → None
Bases: `object`

Namespace / Set of trials

Attributes

description

name

project_id

uid

Methods

compute_uid	

compute_uid(*self*) → str

description = None

name = None

project_id = None

uid

Python Module Index

t

Index

D

E

F

M

N

O

P

Q

R

S

T

U

V

W