RAY

RAY Session

Presented by: Saad

What to cover today

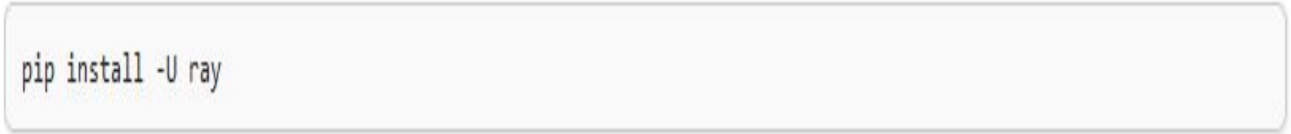
RAY

Gentle Introduction for RAY

- Ray provide Simple, Universal API for building **distributed application**
  - Providing simple primitives for building and running distributed applications.
  - Enabling end users to parallelize single machine code, with little to zero code changes
  - Including a large libraries and tools on to of core Ray to enable complex application
    - Tune: Scalable Hyperparameter Tuning
    - RLlib: Scalable Reinforcement Learning
    - Ray Data processing (Datasets): distributed Arrow on Ray

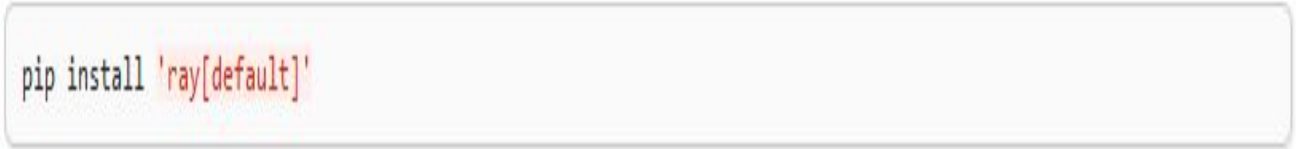
Installing Ray

- Ray currently supports MacOS and Linux. Windows wheels are now available, but Windows support is experimental and under development.



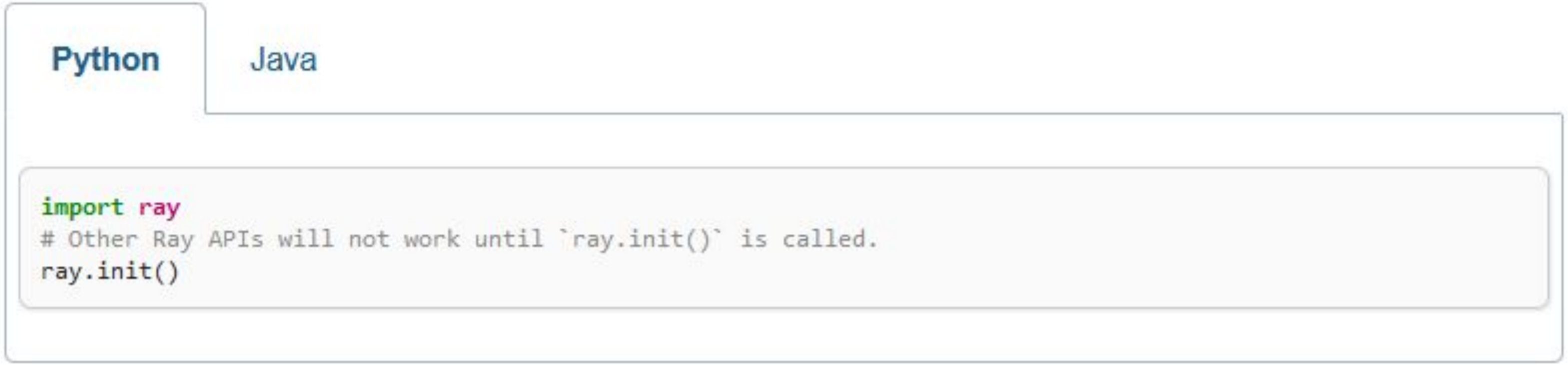
```
pip install -U ray
```

- For Dashboard



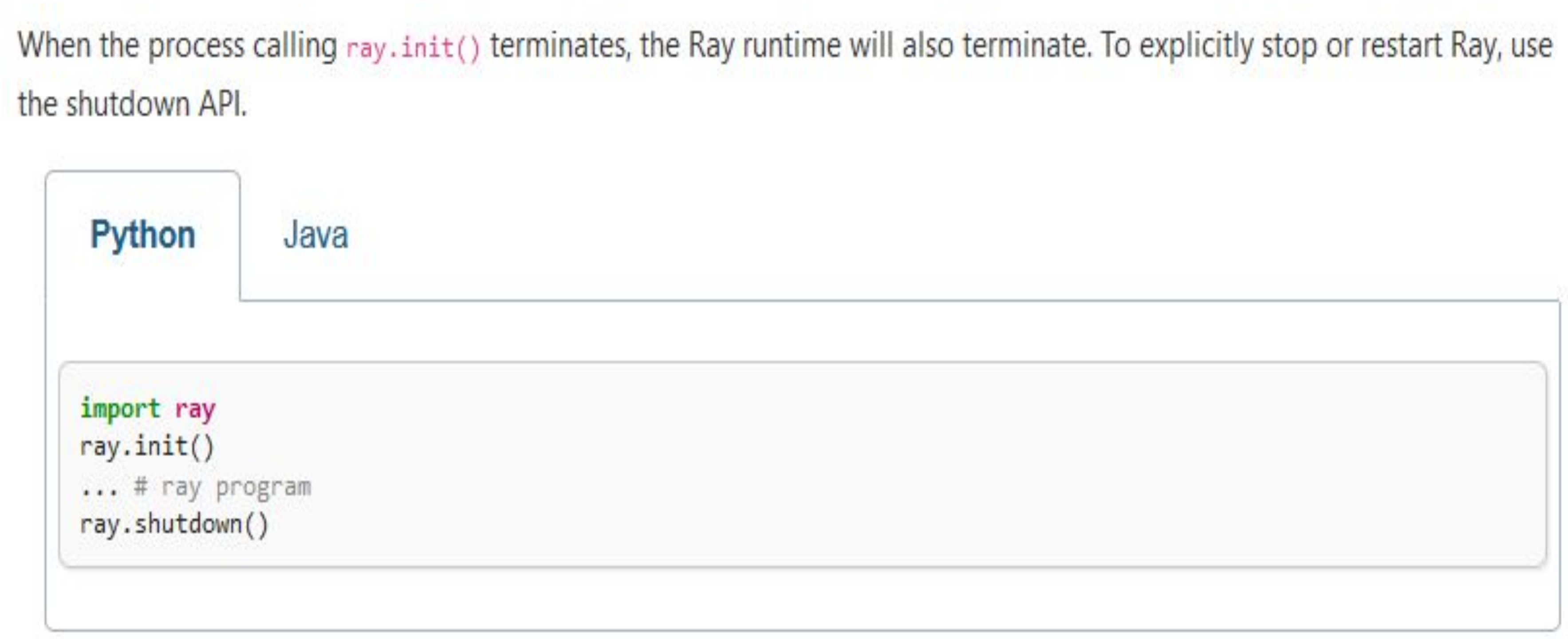
```
pip install 'ray[default]'
```

Starting Ray on a single machine



```
import ray
# Other Ray APIs will not work until `ray.init()` is called.
ray.init()
```

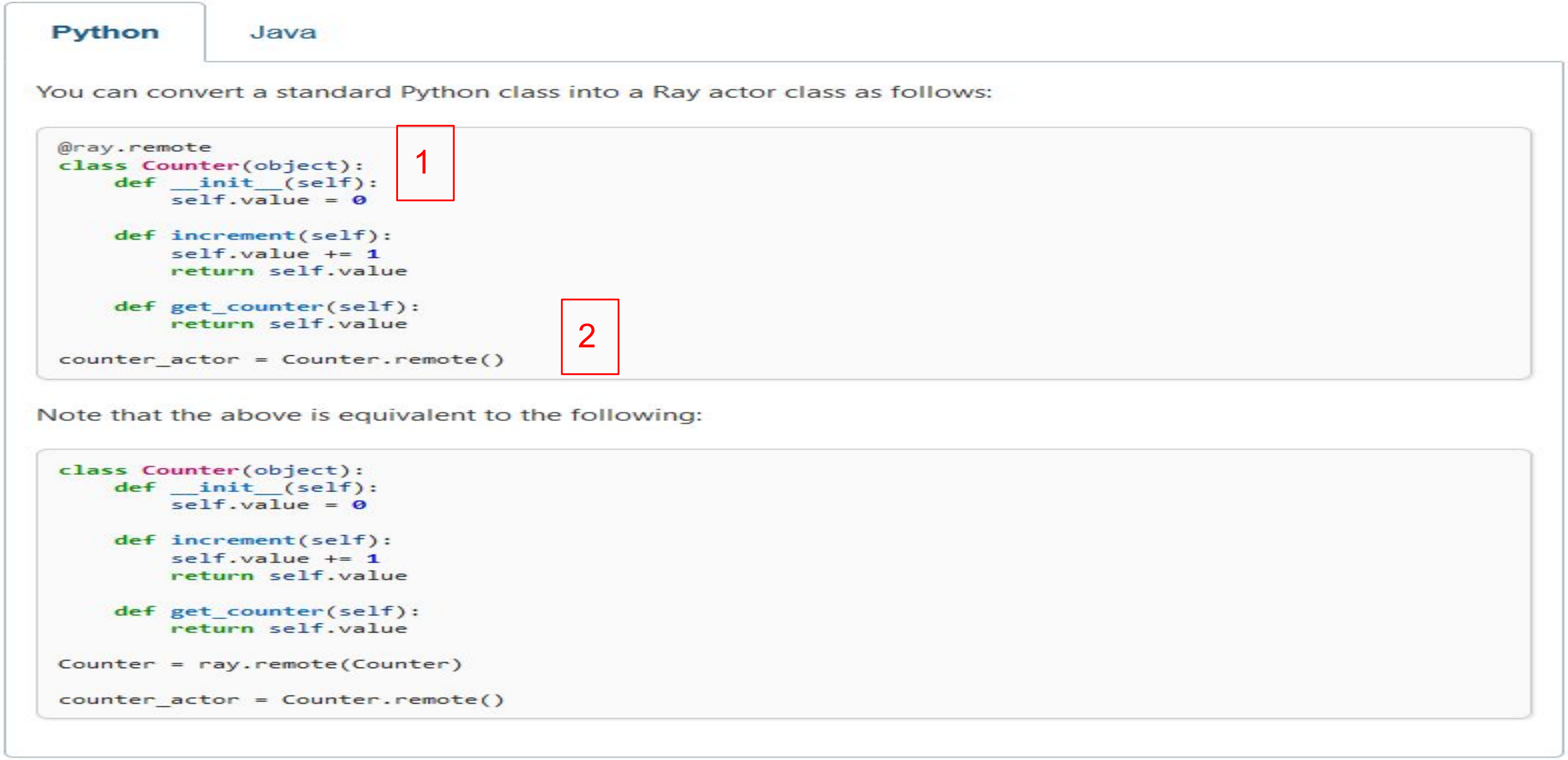
Does it terminates automatically? YES



When the process calling `ray.init()` terminates, the Ray runtime will also terminate. To explicitly stop or restart Ray, use the shutdown API.

```python
import ray
ray.init()
... # ray program
ray.shutdown()
```

Ray Actors (threads, services, worker concept)



Python Java

You can convert a standard Python class into a Ray actor class as follows:

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value

    def get_counter(self):
        return self.value

counter_actor = Counter.remote()
```

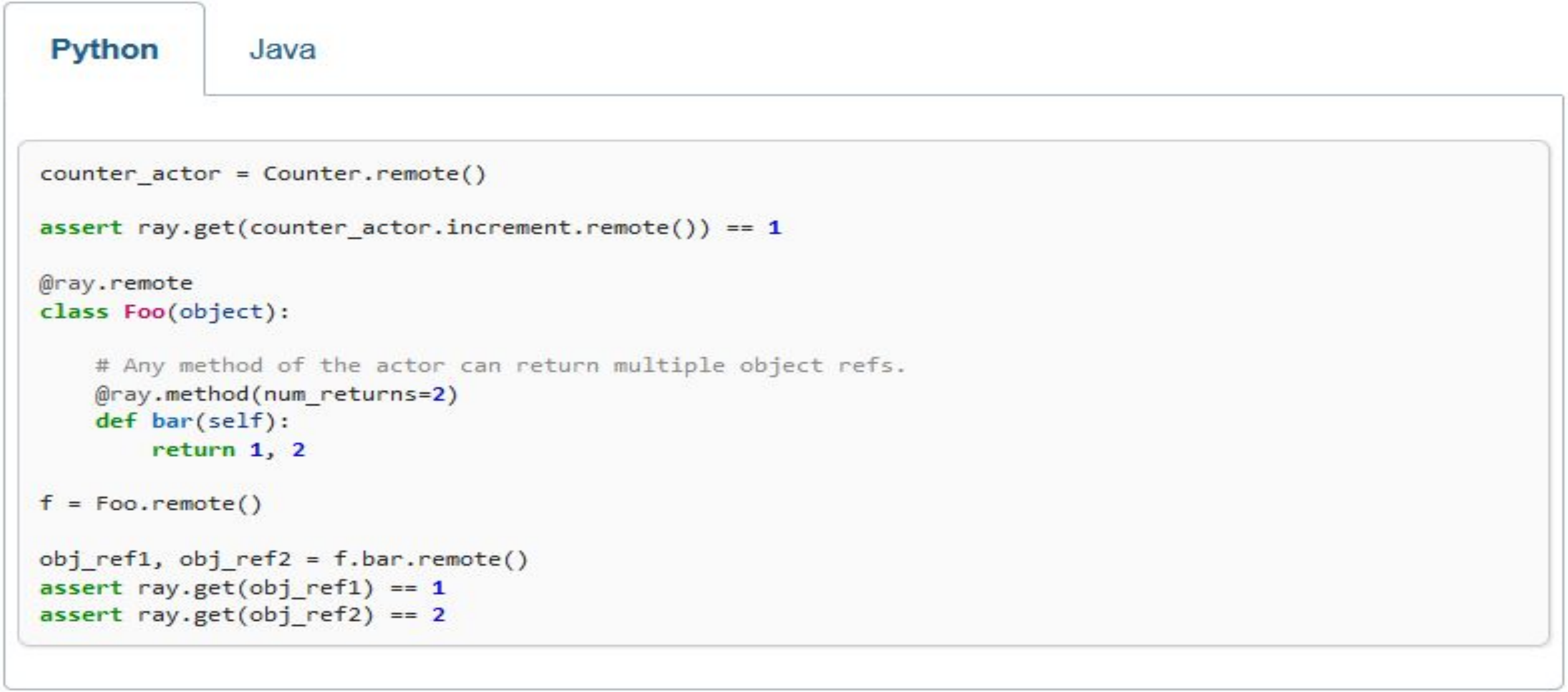
1

2

Note that the above is equivalent to the following:

```
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value

    def get_counter(self):
        return self.value

Counter = ray.remote(Counter)

counter_actor = Counter.remote()
```

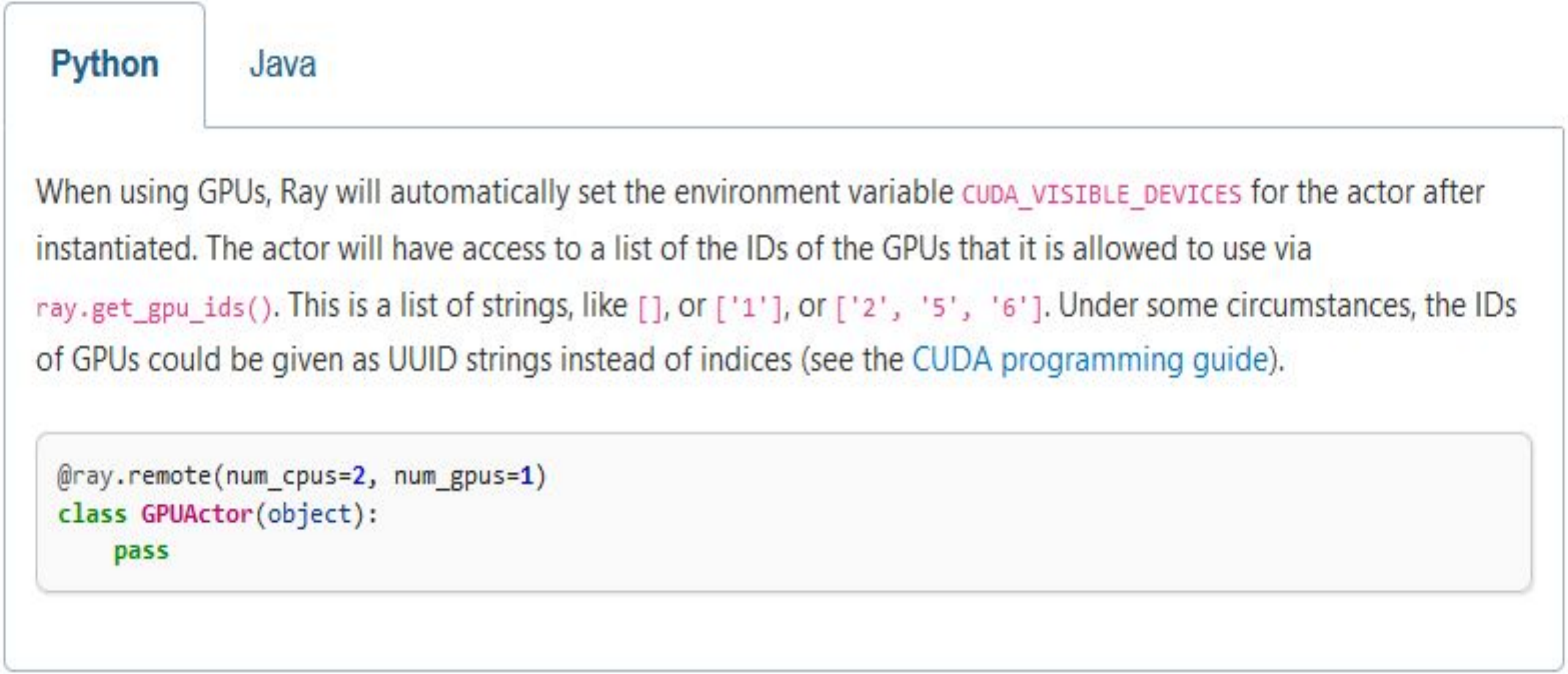
Calling functions inside class



Python Java

```python
counter_actor = Counter.remote()

assert ray.get(counter_actor.increment.remote()) == 1

@ray.remote
class Foo(object):

    # Any method of the actor can return multiple object refs.
    @ray.method(num_returns=2)
    def bar(self):
        return 1, 2

f = Foo.remote()

obj_ref1, obj_ref2 = f.bar.remote()
assert ray.get(obj_ref1) == 1
assert ray.get(obj_ref2) == 2
```

Resources with Actors

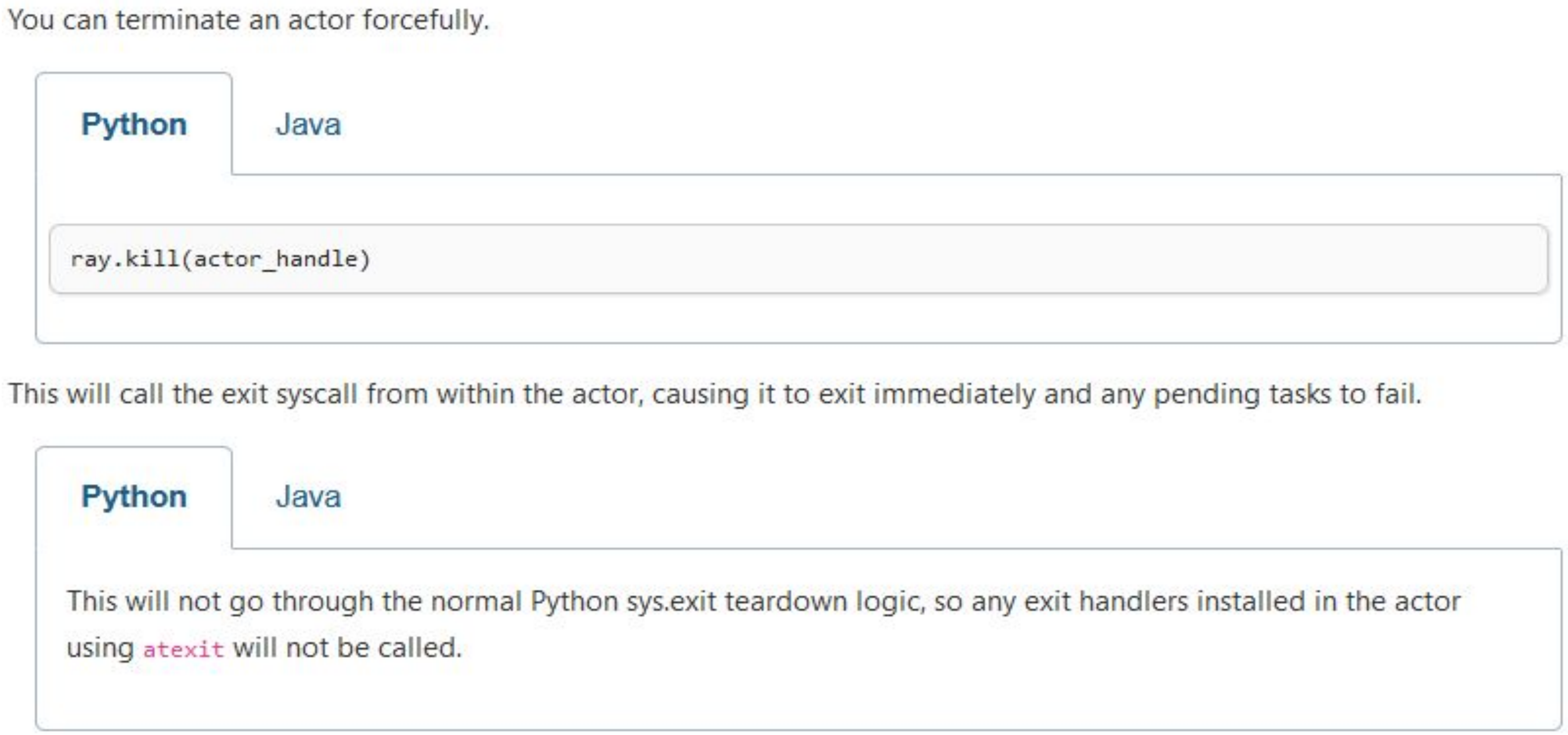


Python Java

When using GPUs, Ray will automatically set the environment variable `CUDA_VISIBLE_DEVICES` for the actor after instantiated. The actor will have access to a list of the IDs of the GPUs that it is allowed to use via `ray.get_gpu_ids()`. This is a list of strings, like `[]`, or `['1']`, or `['2', '5', '6']`. Under some circumstances, the IDs of GPUs could be given as UUID strings instead of indices (see the CUDA programming guide).

```
@ray.remote(num_cpus=2, num_gpus=1)
class GPUActor(object):
    pass
```

Terminate an actor



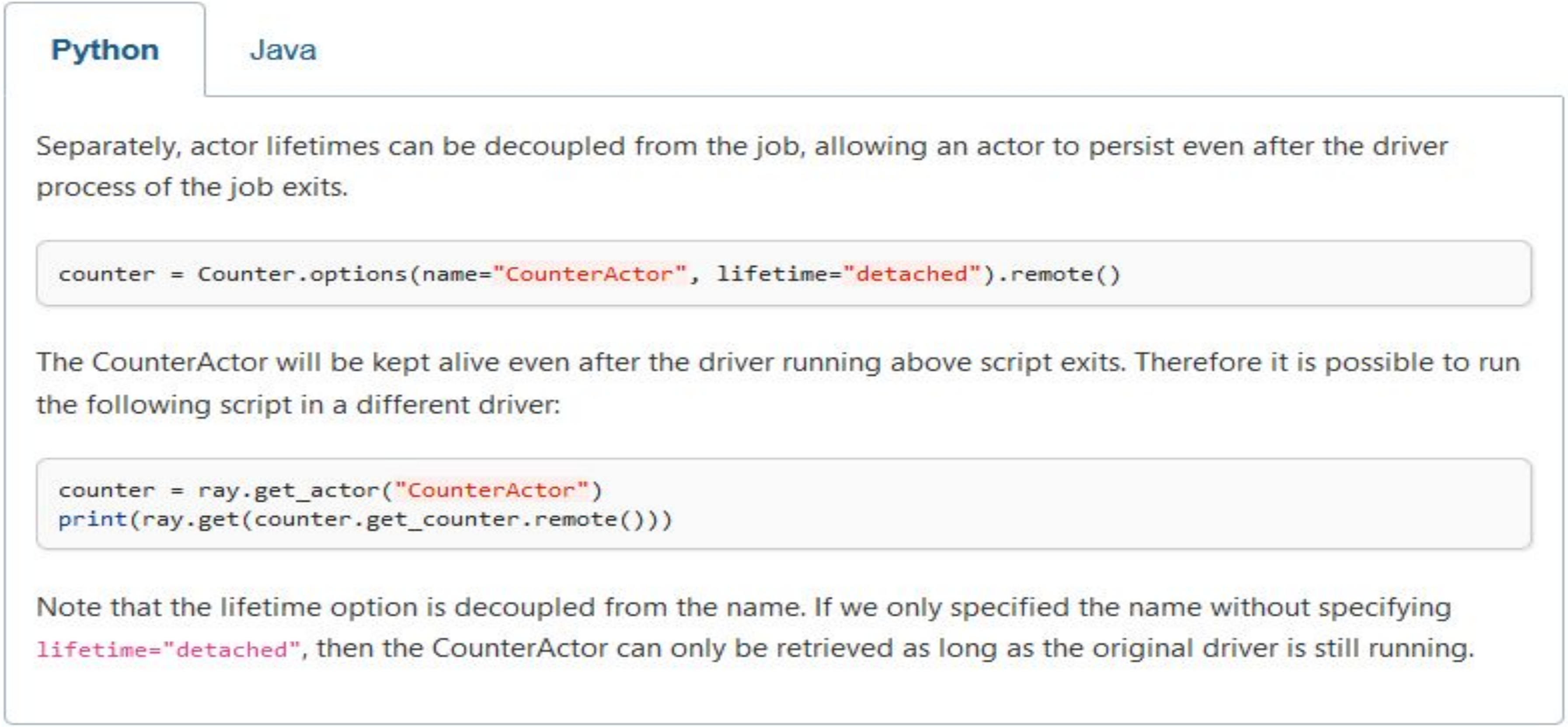
You can terminate an actor forcefully.

```
ray.kill(actor_handle)
```

This will call the exit syscall from within the actor, causing it to exit immediately and any pending tasks to fail.

This will not go through the normal Python sys.exit teardown logic, so any exit handlers installed in the actor using `atexit` will not be called.

Actor Lifetime



Separately, actor lifetimes can be decoupled from the job, allowing an actor to persist even after the driver process of the job exits.

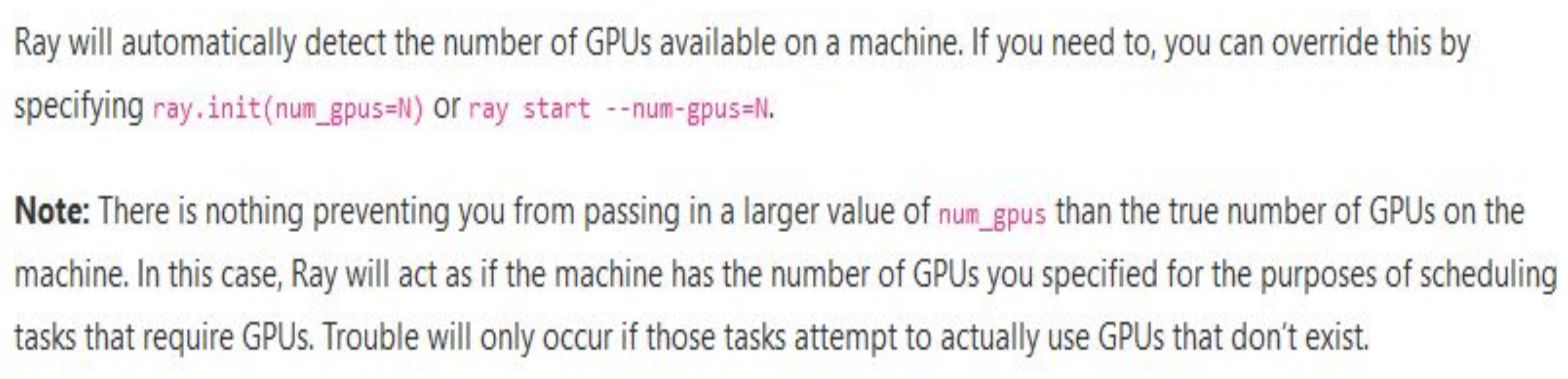
```
counter = Counter.options(name="CounterActor", lifetime="detached").remote()
```

The CounterActor will be kept alive even after the driver running above script exits. Therefore it is possible to run the following script in a different driver:

```
counter = ray.get_actor("CounterActor")
print(ray.get(counter.get_counter.remote()))
```

Note that the lifetime option is decoupled from the name. If we only specified the name without specifying `lifetime="detached"`, then the CounterActor can only be retrieved as long as the original driver is still running.

GPU Support



Ray will automatically detect the number of GPUs available on a machine. If you need to, you can override this by specifying `ray.init(num_gpus=N)` or `ray start --num-gpus=N`.

Note: There is nothing preventing you from passing in a larger value of `num_gpus` than the true number of GPUs on the machine. In this case, Ray will act as if the machine has the number of GPUs you specified for the purposes of scheduling tasks that require GPUs. Trouble will only occur if those tasks attempt to actually use GPUs that don't exist.

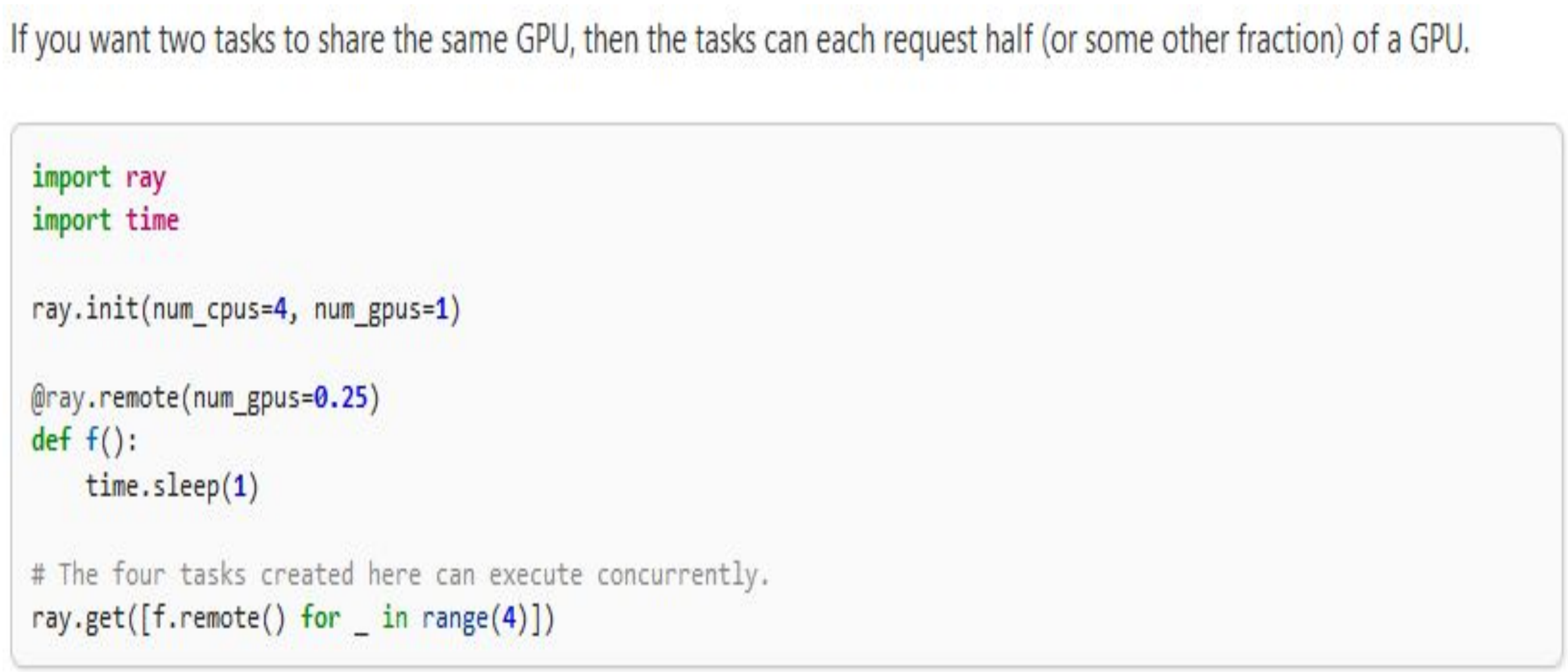
Example

If a remote function requires GPUs, indicate the number of required GPUs in the remote decorator.

```
import os

@ray.remote(num_gpus=1)
def use_gpu():
    print("ray.get_gpu_ids(): {}".format(ray.get_gpu_ids()))
    print("CUDA_VISIBLE_DEVICES: {}".format(os.environ["CUDA_VISIBLE_DEVICES"]))
```

Note: The function `use_gpu` defined above doesn't actually use any GPUs. Ray will schedule it on a machine which has at least one GPU, and will reserve one GPU for it while it is being executed, however it is up to the function to actually make use of the GPU. This is typically done through an external library like TensorFlow. Here is an example that actually uses GPUs. Note that for this example to work, you will need to install the GPU version of TensorFlow.

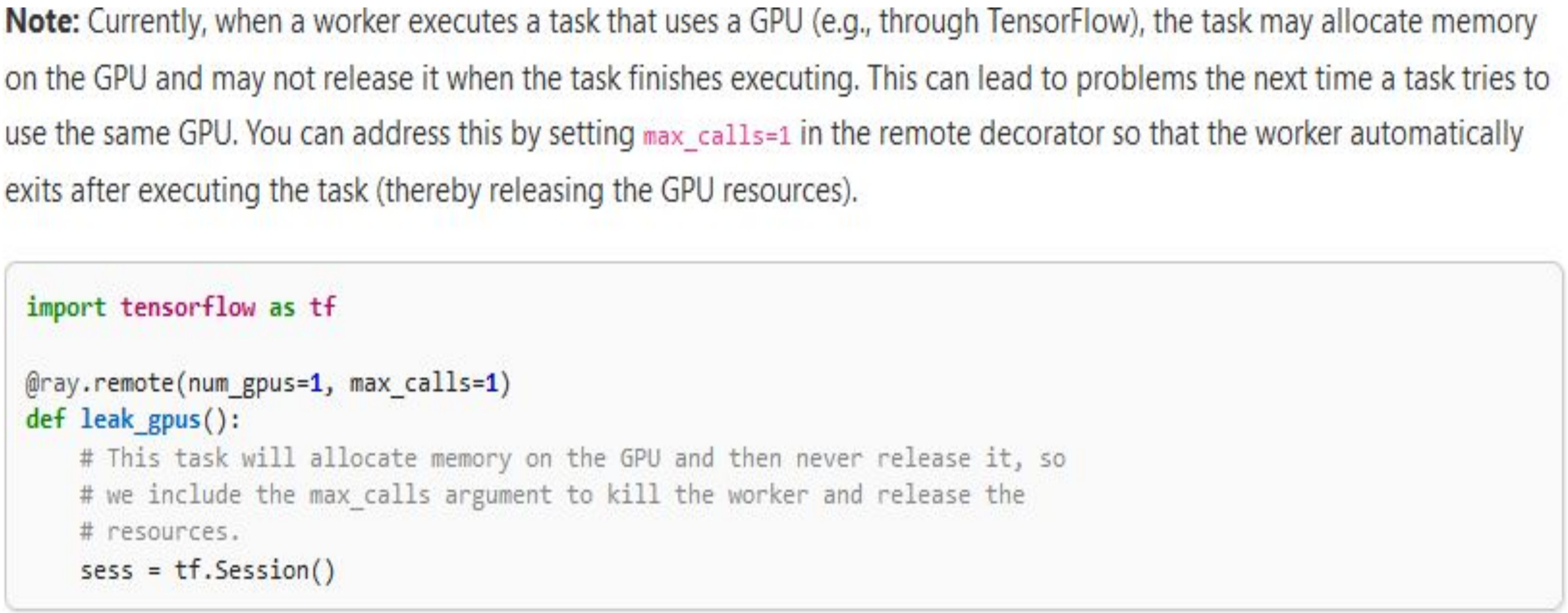
Fractional GPUs



If you want two tasks to share the same GPU, then the tasks can each request half (or some other fraction) of a GPU.

```python
import ray
import time

ray.init(num_cpus=4, num_gpus=1)

@ray.remote(num_gpus=0.25)
def f():
    time.sleep(1)

# The four tasks created here can execute concurrently.
ray.get([f.remote() for _ in range(4)])
```

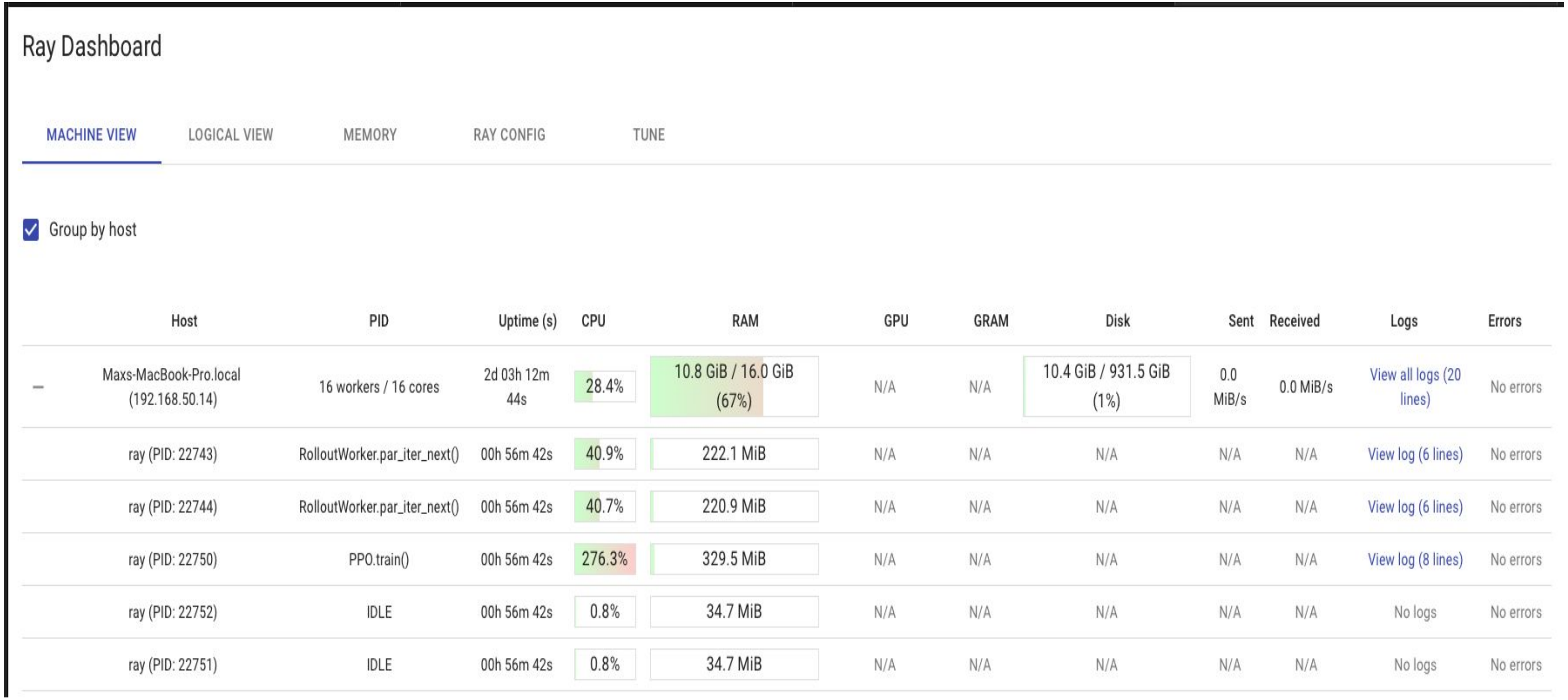
Workers may not Releasing GPU Resources



Note: Currently, when a worker executes a task that uses a GPU (e.g., through TensorFlow), the task may allocate memory on the GPU and may not release it when the task finishes executing. This can lead to problems the next time a task tries to use the same GPU. You can address this by setting `max_calls=1` in the remote decorator so that the worker automatically exits after executing the task (thereby releasing the GPU resources).

```python
import tensorflow as tf

@ray.remote(num_gpus=1, max_calls=1)
def leak_gpus():
    # This task will allocate memory on the GPU and then never release it, so
    # we include the max_calls argument to kill the worker and release the
    # resources.
    sess = tf.Session()
```

Ray Dashboard



Ray Dashboard

☑ Group by host

	Host	PID	Uptime (s)	CPU	RAM	GPU	GRAM	Disk	Sent	Received	Logs	Errors
–	Maxs-MacBook-Pro.local (192.168.50.14)	16 workers / 16 cores	2d 03h 12m 44s	28.4%	10.8 GiB / 16.0 GiB (67%)	N/A	N/A	10.4 GiB / 931.5 GiB (1%)	0.0 MiB/s	0.0 MiB/s	View all logs (20 lines)	No errors
	ray (PID: 22743)	RolloutWorker.par_iter_next()	00h 56m 42s	40.9%	222.1 MiB	N/A	N/A	N/A	N/A	N/A	View log (6 lines)	No errors
	ray (PID: 22744)	RolloutWorker.par_iter_next()	00h 56m 42s	40.7%	220.9 MiB	N/A	N/A	N/A	N/A	N/A	View log (6 lines)	No errors
	ray (PID: 22750)	PPO.train()	00h 56m 42s	276.3%	329.5 MiB	N/A	N/A	N/A	N/A	N/A	View log (8 lines)	No errors
	ray (PID: 22752)	IDLE	00h 56m 42s	0.8%	34.7 MiB	N/A	N/A	N/A	N/A	N/A	No logs	No errors
	ray (PID: 22751)	IDLE	00h 56m 42s	0.8%	34.7 MiB	N/A	N/A	N/A	N/A	N/A	No logs	No errors

Tutorial time on Machine3

See you next session! Hope you Enjoy it