cmd$_{queue}$Documentation

Release 0.1.18

Kitware Inc. Jon Crall

Aug 09, 2023

CONTENTS

1 cmd_queue package **9**
- 1.1 Subpackages . . . 9
  - 1.1.1 cmd_queue.util package . . . 9
    - 1.1.1.1 Submodules . . . 9
      - 1.1.1.1.1 cmd_queue.util.richer module . . . 9
      - 1.1.1.1.2 cmd_queue.util.texter module . . . 9
      - 1.1.1.1.3 cmd_queue.util.textual_extensions module . . . 9
      - 1.1.1.1.4 cmd_queue.util.util_algo module . . . 11
      - 1.1.1.1.5 cmd_queue.util.util_networkx module . . . 12
      - 1.1.1.1.6 cmd_queue.util.util_tags module . . . 12
      - 1.1.1.1.7 cmd_queue.util.util_tmux module . . . 12
      - 1.1.1.1.8 cmd_queue.util.util_yaml module . . . 13
    - 1.1.1.2 Module contents . . . 16
- 1.2 Submodules . . . 16
  - 1.2.1 cmd_queue.__main__ module . . . 16
  - 1.2.2 cmd_queue.airflow_queue module . . . 16
  - 1.2.3 cmd_queue.base_queue module . . . 18
  - 1.2.4 cmd_queue.cli_boilerplate module . . . 20
  - 1.2.5 cmd_queue.monitor_app module . . . 23
  - 1.2.6 cmd_queue.serial_queue module . . . 24
  - 1.2.7 cmd_queue.slurm_queue module . . . 29
  - 1.2.8 cmd_queue.tmux_queue module . . . 32
- 1.3 Module contents . . . 41

2 Indices and tables **51**

Bibliography **53**

Python Module Index **55**

Index **57**

The cmd_queue module is a tool that lets users define a DAG of bash commands. This DAG can be executed in a lightweight tmux backend, or a heavyweight slurm backend, or in simple serial mode that runs in the foreground thread. Rich provides monitoring / live control. For more information see the gitlab README. There is also a Google slides presentation that gives a high level overview.

The following examples show how to use the cmd_queue API in Python. For examples of the Bash API see: *cmd_queue.__main__*.

Example

```
>>> # The available backends classmethod lets you know which backends
>>> # your system has access to. The "serial" backend should always be
>>> # available. Everything else requires some degree of setup (tmux
>>> # is the easiest, just install it, no configuration needed).
>>> import cmd_queue
>>> print(cmd_queue.Queue.available_backends())  # xdoctest: +IGNORE_WANT
['serial', 'tmux', 'slurm']
```

Example

```
>>> # The API to submit jobs is the same regardless of the backend.
>>> # Job dependencies can be specified by name, or by the returned
>>> # job objects.
>>> import cmd_queue
>>> queue = cmd_queue.Queue.create(backend='serial')
>>> job1a = queue.submit('echo "Hello World" && sleep 0.1', name='job1a')
>>> job1b = queue.submit('echo "Hello Revocable" && sleep 0.1', name='job1b')
>>> job2a = queue.submit('echo "Hello Crushed" && sleep 0.1', depends=[job1a], name=
↪'job2a')
>>> job2b = queue.submit('echo "Hello Shadow" && sleep 0.1', depends=[job1b], name='job2b
↪')
>>> job3 = queue.submit('echo "Hello Excavate" && sleep 0.1', depends=[job2a, job2b],␣
↪name='job3')
>>> jobX = queue.submit('echo "Hello Barrette" && sleep 0.1', depends=[], name='jobX')
>>> jobY = queue.submit('echo "Hello Overwrite" && sleep 0.1', depends=[jobX], name='jobY
↪')
>>> jobZ = queue.submit('echo "Hello Giblet" && sleep 0.1', depends=[jobY], name='jobZ')
...
>>> # Use print_graph to get a "network text" representation of the DAG
>>> # This gives you a sense of what jobs can run in parallel
>>> queue.print_graph(reduced=False)
Graph:
╾ job1a
    └─ job2a
        └─ job3  job2b
╾ job1b
    └─ job2b
        └─  ...
╾ jobX
    └─ jobY
        └─ jobZ
```

(continues on next page)

(continued from previous page)

```
>>> # The purpose of command queue is not to run the code, but to
>>> # generate the code that would run the code.
>>> # The print_commands method gives you the gist of the code
>>> # command queue would run. Flags can be given to modify conciseness.
>>> queue.print_commands(style='plain')
# --- ...
#!/bin/bash
# Written by cmd_queue ...

# ----
# Jobs
# ----

#
### Command 1 / 8 - job1a
echo "Hello World" && sleep 0.1
#
### Command 2 / 8 - job1b
echo "Hello Revocable" && sleep 0.1
#
### Command 3 / 8 - job2a
echo "Hello Crushed" && sleep 0.1
#
### Command 4 / 8 - job2b
echo "Hello Shadow" && sleep 0.1
#
### Command 5 / 8 - job3
echo "Hello Excavate" && sleep 0.1
#
### Command 6 / 8 - jobX
echo "Hello Barrette" && sleep 0.1
#
### Command 7 / 8 - jobY
echo "Hello Overwrite" && sleep 0.1
#
### Command 8 / 8 - jobZ
echo "Hello Giblet" && sleep 0.1
>>> # Different backends have different ways of executing the
>>> # the underlying DAG, but it always boils down to: generate the code
>>> # that would execute your jobs.
>>> #
>>> # For the TMUX queue it boils down to writing a bash script for
>>> # sessions that can run in parallel, and a bash script that submits
>>> # them as different sessions (note: locks exist but are omitted here)
>>> tmux_queue = queue.change_backend('tmux', size=2)
>>> tmux_queue.print_commands(style='plain', with_locks=0)
# --- ...sh
#!/bin/bash
# Written by cmd_queue ...
# ----
# Jobs
# ----
```

(continues on next page)

(continued from previous page)

```
#
### Command 1 / 3 - jobX
echo "Hello Barrette" && sleep 0.1
#
### Command 2 / 3 - jobY
echo "Hello Overwrite" && sleep 0.1
#
### Command 3 / 3 - jobZ
echo "Hello Giblet" && sleep 0.1
# --- ...sh
#!/bin/bash
# Written by cmd_queue ...
# ----
# Jobs
# ----
#
### Command 1 / 4 - job1a
echo "Hello World" && sleep 0.1
#
### Command 2 / 4 - job2a
echo "Hello Crushed" && sleep 0.1
#
### Command 3 / 4 - job1b
echo "Hello Revocable" && sleep 0.1
#
### Command 4 / 4 - job2b
echo "Hello Shadow" && sleep 0.1
# --- ...sh
#!/bin/bash
# Written by cmd_queue ...
# ----
# Jobs
# ----
#
### Command 1 / 1 - job3
echo "Hello Excavate" && sleep 0.1
# --- ...sh
#!/bin/bash
# Driver script to start the tmux-queue
echo "Submitting 8 jobs to a tmux queue"
### Run Queue: cmdq_unnamed_000_... with 3 jobs
tmux new-session -d -s cmdq_unnamed_000_... "bash"
tmux send -t cmdq_unnamed_... \
    "source ...sh" \
    Enter
### Run Queue: cmdq_unnamed_001_... with 4 jobs
tmux new-session -d -s cmdq_unnamed_001_... "bash"
tmux send -t cmdq_unnamed_001_... \
    "source ...sh" \
    Enter
### Run Queue: cmdq_unnamed_002_... with 1 jobs
tmux new-session -d -s cmdq_unnamed_002_... "bash"
```

(continues on next page)

(continued from previous page)

```
tmux send -t cmdq_unnamed_... \
    "source ...sh" \
    Enter
echo "Spread jobs across 3 tmux workers"
>>> # The slurm queue is very simple, it just constructs one bash file that is the
>>> # sbatch commands to submit your jobs. All of the other details are taken care of
>>> # by slurm itself.
>>> # xdoctest: +IGNORE_WANT
>>> slurm_queue = queue.change_backend('slurm')
>>> slurm_queue.print_commands(style='plain')
# --- ...sh
mkdir -p ".../logs"
JOB_000=$(sbatch --job-name="job1a" --output="/.../logs/job1a.sh" --wrap 'echo "Hello␣
↪World" && sleep 0.1' --parsable)
JOB_001=$(sbatch --job-name="job1b" --output="/.../logs/job1b.sh" --wrap 'echo "Hello␣
↪Revocable" && sleep 0.1' --parsable)
JOB_002=$(sbatch --job-name="jobX" --output="/.../logs/jobX.sh" --wrap 'echo "Hello␣
↪Barrette" && sleep 0.1' --parsable)
JOB_003=$(sbatch --job-name="job2a" --output="/.../logs/job2a.sh" --wrap 'echo "Hello␣
↪Crushed" && sleep 0.1' "--dependency=afterok:${JOB_000}" --parsable)
JOB_004=$(sbatch --job-name="job2b" --output="/.../logs/job2b.sh" --wrap 'echo "Hello␣
↪Shadow" && sleep 0.1' "--dependency=afterok:${JOB_001}" --parsable)
JOB_005=$(sbatch --job-name="jobY" --output="/.../logs/jobY.sh" --wrap 'echo "Hello␣
↪Overwrite" && sleep 0.1' "--dependency=afterok:${JOB_002}" --parsable)
JOB_006=$(sbatch --job-name="job3" --output="/.../logs/job3.sh" --wrap 'echo "Hello␣
↪Excavate" && sleep 0.1' "--dependency=afterok:${JOB_003}:${JOB_004}" --parsable)
JOB_007=$(sbatch --job-name="jobZ" --output="/.../logs/jobZ.sh" --wrap 'echo "Hello␣
↪Giblet" && sleep 0.1' "--dependency=afterok:${JOB_005}" --parsable)
>>> # The airflow backend is slightly different because it defines
>>> # DAGs with Python files, so we write a Python file instead of
>>> # a bash file. NOTE: the process of actually executing the airflow
>>> # DAG has not been finalized yet. (Help wanted)
>>> airflow_queue = queue.change_backend('airflow')
>>> airflow_queue.print_commands(style='plain')
# --- ...py
from airflow import DAG
from datetime import timezone
from datetime import datetime as datetime_cls
from airflow.operators.bash import BashOperator
now = datetime_cls.utcnow().replace(tzinfo=timezone.utc)
dag = DAG(
    'SQ',
    start_date=now,
    catchup=False,
    tags=['example'],
)
jobs = dict()
jobs['job1a'] = BashOperator(task_id='job1a', bash_command='echo "Hello World" && sleep␣
↪0.1', dag=dag)
jobs['job1b'] = BashOperator(task_id='job1b', bash_command='echo "Hello Revocable" &&␣
↪sleep 0.1', dag=dag)
jobs['job2a'] = BashOperator(task_id='job2a', bash_command='echo "Hello Crushed" &&␣
```

(continues on next page)

(continued from previous page)

```
↪sleep 0.1', dag=dag)
jobs['job2b'] = BashOperator(task_id='job2b', bash_command='echo "Hello Shadow" && sleep␣
↪0.1', dag=dag)
jobs['job3'] = BashOperator(task_id='job3', bash_command='echo "Hello Excavate" && sleep␣
↪0.1', dag=dag)
jobs['jobX'] = BashOperator(task_id='jobX', bash_command='echo "Hello Barrette" && sleep␣
↪0.1', dag=dag)
jobs['jobY'] = BashOperator(task_id='jobY', bash_command='echo "Hello Overwrite" &&␣
↪sleep 0.1', dag=dag)
jobs['jobZ'] = BashOperator(task_id='jobZ', bash_command='echo "Hello Giblet" && sleep 0.
↪1', dag=dag)
jobs['job2a'].set_upstream(jobs['job1a'])
jobs['job2b'].set_upstream(jobs['job1b'])
jobs['job3'].set_upstream(jobs['job2a'])
jobs['job3'].set_upstream(jobs['job2b'])
jobs['jobY'].set_upstream(jobs['jobX'])
jobs['jobZ'].set_upstream(jobs['jobY'])
```

Example

```
>>> # Given a Queue object, the "run" method will attempt to execute it
>>> # for you and give you a sense of progress.
>>> # xdoctest: +IGNORE_WANT
>>> import cmd_queue
>>> queue = cmd_queue.Queue.create(backend='serial')
>>> job1a = queue.submit('echo "Hello World" && sleep 0.1', name='job1a')
>>> job1b = queue.submit('echo "Hello Revocable" && sleep 0.1', name='job1b')
>>> job2a = queue.submit('echo "Hello Crushed" && sleep 0.1', depends=[job1a], name=
↪'job2a')
>>> job2b = queue.submit('echo "Hello Shadow" && sleep 0.1', depends=[job1b], name='job2b
↪')
>>> job3 = queue.submit('echo "Hello Excavate" && sleep 0.1', depends=[job2a, job2b],␣
↪name='job3')
>>> jobX = queue.submit('echo "Hello Barrette" && sleep 0.1', depends=[], name='jobX')
>>> jobY = queue.submit('echo "Hello Overwrite" && sleep 0.1', depends=[jobX], name='jobY
↪')
>>> jobZ = queue.submit('echo "Hello Giblet" && sleep 0.1', depends=[jobY], name='jobZ')
>>> # Using the serial queue simply executes all of the commands in order in
>>> # the current session. This behavior can be useful as a fallback or
>>> # for debugging.
>>> # Note: xdoctest doesnt seem to capture the set -x parts. Not sure why.
>>> queue.run(block=True, system=True)  # xdoctest: +IGNORE_WANT
─── START CMD ───
[ubelt.cmd] ...@...:...$ bash ...sh
+ echo 'Hello World'
Hello World
+ sleep 0.1
+ echo 'Hello Revocable'
Hello Revocable
+ sleep 0.1
```

(continues on next page)

(continued from previous page)

```
+ echo 'Hello Crushed'
Hello Crushed
+ sleep 0.1
+ echo 'Hello Shadow'
Hello Shadow
+ sleep 0.1
+ echo 'Hello Excavate'
Hello Excavate
+ sleep 0.1
+ echo 'Hello Barrette'
Hello Barrette
+ sleep 0.1
+ echo 'Hello Overwrite'
Hello Overwrite
+ sleep 0.1
+ echo 'Hello Giblet'
Hello Giblet
+ sleep 0.1
Command Queue Final Status:
{"status": "done", "passed": 8, "failed": 0, "skipped": 0, "total": 8, "name": "",
↪"rootid": "..."}
└─── END CMD ───
>>> # The TMUX queue does not show output directly by default (although
>>> # it does have access to methods that let it grab logs from tmux)
>>> # But normally you can attach to the tmux sessions to look at them
>>> # The default monitor will depend on if you have textual installed or not.
>>> # Another default behavior is that it will ask if you want to kill
>>> # previous command queue tmux sessions, but this can be disabled.
>>> import ubelt as ub
>>> if 'tmux' in cmd_queue.Queue.available_backends():
>>>     tmux_queue = queue.change_backend('tmux', size=2)
>>>     tmux_queue.run(with_textual='auto', check_other_sessions=False)
[ubelt.cmd] joncrall@calculex:~/code/cmd_queue$ bash /home/joncrall/.cache/cmd_queue/
↪tmux/unnamed_2022-07-27_cbfeedda/run_queues_unnamed.sh
submitting 8 jobs
jobs submitted

 tmux session name  status  passed  failed  skipped  total

┌ cmdq_unnamed_000  done    3       0       0        3     ┐
  cmdq_unnamed_001  done    4       0       0        4
  cmdq_unnamed_002  done    1       0       0        1
  agg               done    8       0       0        8     ┘
>>> # The monitoring for the slurm queue is basic, and the extent to
>>> # which features can be added will depend on your slurm config.
>>> # Any other slurm monitoring tools can be used. There are plans
>>> # to implement a textual monitor based on the slurm logfiles.
>>> if 'slurm' in cmd_queue.Queue.available_backends():
>>>     slurm_queue = queue.change_backend('slurm')
>>>     slurm_queue.run()
─── START CMD ───
```

(continues on next page)

(continued from previous page)

```
[ubelt.cmd] ...sh
└─── END CMD ───
                        slurm-monitor

 num_running  num_in_queue  total_monitored  num_at_start

┌ 0           │ 31           │ 118              │ 118          │
└─────────────────────────────────────────────────────────────────
>>> # xdoctest: +SKIP
>>> # Running airflow queues is not implemented yet
>>> if 'airflow' in cmd_queue.Queue.available_backends():
>>>     airflow_queue = queue.change_backend('airflow')
>>>     airflow_queue.run()
```

CHAPTER

ONE

CMD_QUEUE PACKAGE

1.1 Subpackages

1.1.1 cmd_queue.util package

1.1.1.1 Submodules

1.1.1.1.1 cmd_queue.util.richer module

An automatic lazy rich API

Example

from cmd_queue.util import richer as rich

1.1.1.1.2 cmd_queue.util.texter module

An automatic lazy textual API

Example

from cmd_queue.util import texter

1.1.1.1.3 cmd_queue.util.textual_extensions module

class cmd_queue.util.textual_extensions.**class_or_instancemethod**

Bases: classmethod

Allows a method to behave as a class or instance method

References

https://stackoverflow.com/questions/28237955/same-name-for-classmethod-and-instancemethod

Example

```
>>> class X:
...     @class_or_instancemethod
...     def foo(self_or_cls):
...         if isinstance(self_or_cls, type):
...             return f"bound to the class"
...         else:
...             return f"bound to the instance"
>>> print(X.foo())
bound to the class
>>> print(X().foo())
bound to the instance
```

class cmd_queue.util.textual_extensions.**InstanceRunnableApp**

Bases: object

Extension of App that allows for running an instance

CommandLine

```
xdoctest -m cmd_queue.util.textual_extensions InstanceRunnableApp:0 --interact
```

Example

```
>>> # xdoctest: +REQUIRES(module:textual)
>>> # xdoctest: +REQUIRES(--interact)
>>> from textual import events
>>> #from textual.widgets import ScrollView
>>> from textual.scroll_view import ScrollView
>>> class DemoApp(InstanceRunnableApp):
>>>     def __init__(self, myvar, **kwargs):
>>>         super().__init__(**kwargs)
>>>         self.myvar = myvar
>>>     async def on_load(self, event: events.Load) -> None:
>>>         await self.bind("q", "quit", "Quit")
>>>     async def on_mount(self, event: events.Mount) -> None:
>>>         self.body = body = ScrollView(auto_width=True)
>>>         await self.view.dock(body)
>>>         async def add_content():
>>>             from rich.text import Text
>>>             content = Text(self.myvar)
>>>             await body.update(content)
>>>         await self.call_later(add_content)
>>> DemoApp.run(myvar='Existing classmethod way of running an App')
```

(continues on next page)

(continued from previous page)

```
>>> self = DemoApp(myvar='The instance way of running an App')
>>> self.run()
```

classmethod _run_as_cls(*console=None*, *screen: bool = True*, *driver=None*, ***kwargs*)

Original classmethod logic

_run_as_instance(*console=None*, *screen: bool = True*, *driver=None*, ***kwargs*)

New instancemethod logic

classmethod run(*console=None*, *screen: bool = True*, *driver=None*, ***kwargs*)

Run the app. :Parameters: * **console** (*Console, optional*) – Console object. Defaults to None.

- **screen** (*bool, optional*) – Enable application mode. Defaults to True.
- **driver** (*Type[Driver], optional*) – Driver class or None for default. Defaults to None.

1.1.1.1.4 cmd_queue.util.util_algo module

cmd_queue.util.util_algo.**balanced_number_partitioning**(*items*, *num_parts*)

Greedy approximation to multiway number partitioning

Uses Greedy number partitioning method to minimize the size of the largest partition.

Parameters

- **items** (*np.ndarray*) – list of numbers (i.e. weights) to split between paritions.
- **num_parts** (*int*) – number of partitions

Returns

A list for each parition that contains the index of the items assigned to it.

Return type

List[np.ndarray]

References

https://en.wikipedia.org/wiki/Multiway_number_partitioning https://en.wikipedia.org/wiki/Balanced_number_partitioning

Example

```
>>> from cmd_queue.util.util_algo import balanced_number_partitioning
>>> items = np.array([1, 3, 29, 22, 4, 5, 9])
>>> num_parts = 3
>>> bin_assignments = balanced_number_partitioning(items, num_parts)
>>> # xdoctest: +REQUIRES(module:kwarray)
>>> import kwarray
>>> groups = kwarray.apply_grouping(items, bin_assignments)
>>> bin_weights = [g.sum() for g in groups]
```

1.1.1.1.5 cmd_queue.util.util_networkx module

cmd_queue.util.util_networkx.**is_topological_order**(*graph*, *node_order*)

A topological ordering of nodes is an ordering of the nodes such that for every edge (u,v) in G, u appears earlier than v in the ordering

Runtime:
O(V * E)

References

https://stackoverflow.com/questions/54174116/checking-validity-of-topological-sort

Example

```
>>> import networkx as nx
>>> raw = nx.erdos_renyi_graph(100, 0.5, directed=True, seed=3432)
>>> graph = nx.DiGraph(nodes=raw.nodes())
>>> graph.add_edges_from([(u, v) for u, v in raw.edges() if u < v])
>>> node_order = list(nx.topological_sort(graph))
>>> assert is_topological_order(graph, node_order)
>>> assert not is_topological_order(graph, node_order[::-1])
```

1.1.1.1.6 cmd_queue.util.util_tags module

class cmd_queue.util.util_tags.**Tags**(*iterable=()*, */*)

Bases: list

A glorified List[str] with special extra methods

classmethod coerce(*tags*)

Coerce the tags to a list of strings or None

intersection(*other*)

1.1.1.1.7 cmd_queue.util.util_tmux module

Generic tmux helpers

class cmd_queue.util.util_tmux.**tmux**

Bases: object

static list_sessions()

static _kill_session_command(*target_session*)

static _capture_pane_command(*target_session*)

static capture_pane(*target_session*, *verbose=3*)

static kill_session(*target_session*, *verbose=3*)

1.1.1.1.8 cmd_queue.util.util_yaml module

class cmd_queue.util.util_yaml._YamlRepresenter

Bases: object

static str_presenter(*dumper*, *data*)

cmd_queue.util.util_yaml._custom_ruaml_loader()

References

https://stackoverflow.com/questions/59635900/ruamel-yaml-custom-commentedmapping-for-custom-tags
https://stackoverflow.com/questions/528281/how-can-i-include-a-yaml-file-inside-another

cmd_queue.util.util_yaml._custom_ruaml_dumper()

References

https://stackoverflow.com/questions/59635900/ruamel-yaml-custom-commentedmapping-for-custom-tags

cmd_queue.util.util_yaml._custom_pyaml_dumper()

class cmd_queue.util.util_yaml.Yaml

Bases: object

Namespace for yaml functions

static dumps(*data*, *backend='ruamel'*)

Dump yaml to a string representation (and account for some of our use-cases)

Parameters

- **data** (*Any*) – yaml representable data
- **backend** (*str*) – either ruamel or pyyaml

Returns
yaml text

Return type
str

Example

```
>>> import ubelt as ub
>>> data = {
>>>     'a': 'hello world',
>>>     'b': ub.udict({'a': 3})
>>> }
>>> text1 = Yaml.dumps(data, backend='ruamel')
>>> print(text1)
>>> text2 = Yaml.dumps(data, backend='pyyaml')
>>> print(text2)
>>> assert text1 == text2
```

static load(*file*, *backend='ruamel'*)

Load yaml from a file

Parameters

- **file** (*io.TextIOBase | PathLike | str*) – yaml file path or file object
- **backend** (*str*) – either ruamel or pyyaml

Returns
object

static loads(*text*, *backend='ruamel'*)

Load yaml from a text

Parameters

- **text** (*str*) – yaml text
- **backend** (*str*) – either ruamel or pyyaml

Returns
object

Example

```
>>> import ubelt as ub
>>> data = {
>>>     'a': 'hello world',
>>>     'b': ub.udict({'a': 3})
>>> }
>>> print('data = {}'.format(ub.urepr(data, nl=1)))
>>> print('---')
>>> text = Yaml.dumps(data)
>>> print(ub.highlight_code(text, 'yaml'))
>>> print('---')
>>> data2 = Yaml.loads(text)
>>> assert data == data2
>>> data3 = Yaml.loads(text, backend='pyyaml')
>>> print('data2 = {}'.format(ub.urepr(data2, nl=1)))
>>> print('data3 = {}'.format(ub.urepr(data3, nl=1)))
>>> assert data == data3
```

static coerce(*data*, *backend='ruamel'*)

Attempt to convert input into a parsed yaml / json data structure. If the data looks like a path, it tries to load and parse file contents. If the data looks like a yaml/json string it tries to parse it. If the data looks like parsed data, then it returns it as-is.

Parameters

- **data** (*str | PathLike | dict | list*)
- **backend** (*str*) – either ruamel or pyyaml

Returns
parsed yaml data

Return type
object

Note: The input to the function cannot distinguish a string that should be loaded and a string that should be parsed. If it looks like a file that exists it will read it. To avoid this coerner case use this only for data where you expect the output is a List or Dict.

References

https://stackoverflow.com/questions/528281/how-can-i-include-a-yaml-file-inside-another

Example

```
>>> Yaml.coerce('"[1, 2, 3]"')
[1, 2, 3]
>>> fpath = ub.Path.appdir('cmd_queue/tests/util_yaml').ensuredir() / 'file.yaml
↪'
>>> fpath.write_text(Yaml.dumps([4, 5, 6]))
>>> Yaml.coerce(fpath)
[4, 5, 6]
>>> Yaml.coerce(str(fpath))
[4, 5, 6]
>>> dict(Yaml.coerce('{a: b, c: d}'))
{'a': 'b', 'c': 'd'}
>>> Yaml.coerce(None)
None
```

Example

```
>>> assert Yaml.coerce('') is None
```

Example

```
>>> dpath = ub.Path.appdir('cmd_queue/tests/util_yaml').ensuredir()
>>> fpath = dpath / 'external.yaml'
>>> fpath.write_text(Yaml.dumps({'foo': 'bar'}))
>>> text = ub.codeblock(
>>>     f'''
>>>     items:
>>>         - !include {dpath}/external.yaml
>>>     ''')
>>> data = Yaml.coerce(text, backend='ruamel')
>>> print(Yaml.dumps(data, backend='ruamel'))
items:
- foo: bar
```

```
>>> text = ub.codeblock(
>>>     f'''
>>>     items:
```

(continues on next page)

(continued from previous page)

```
>>>         !include [{dpath}/external.yaml, blah, 1, 2, 3]
>>>     ''')
>>> data = Yaml.coerce(text, backend='ruamel')
>>> print('data = {}'.format(ub.urepr(data, nl=1)))
>>> print(Yaml.dumps(data, backend='ruamel'))
```

static InlineList(*items*)

References

static Dict(*data*)

Get a ruamel-enhanced dictionary

Example

```
>>> data = {'a': 'avalue', 'b': 'bvalue'}
>>> data = Yaml.Dict(data)
>>> data.yaml_set_start_comment('hello')
>>> # Note: not working https://sourceforge.net/p/ruamel-yaml/tickets/400/
>>> data.yaml_set_comment_before_after_key('a', before='a comment', indent=2)
>>> data.yaml_set_comment_before_after_key('b', 'b comment')
>>> print(Yaml.dumps(data))
```

static CodeBlock(*text*)

1.1.1.2 Module contents

1.2 Submodules

1.2.1 cmd_queue.__main__ module

1.2.2 cmd_queue.airflow_queue module

UNFINISHED - NOT FUNCTIONAL

Airflow backend

Requires:

pip install apache-airflow pip install apache-airflow[cncf.kubernetes]

class cmd_queue.airflow_queue.**AirflowJob**(*command*, *name=None*, *output_fpath=None*, *depends=None*, *partition=None*, *cpus=None*, *gpus=None*, *mem=None*, *begin=None*, *shell=None*, ***kwargs*)

Bases: *Job*

Represents a airflow job that hasn't been executed yet

finalize_text()

class cmd_queue.airflow_queue.**AirflowQueue**(*name=None*, *shell=None*, ***kwargs*)

Bases: *Queue*

Example

```
>>> # xdoctest: +REQUIRES(module:airflow)
>>> # xdoctest: +SKIP
>>> from cmd_queue.airflow_queue import *  # NOQA
>>> self = AirflowQueue('zzz_mydemo')
>>> job1 = self.submit('echo hi 1 && true')
>>> job2 = self.submit('echo hi 2 && true')
>>> job3 = self.submit('echo hi 3 && true', depends=job1)
>>> self.print_commands()
>>> self.write()
>>> self.run()
>>> #self.run()
>>> # self.read_state()
```

classmethod is_available()

Determines if we can run the tmux queue or not.

run(*block=True*, *system=False*)

finalize_text()

submit(*command*, ***kwargs*)

print_commands(*with_status=False*, *with_gaurds=False*, *with_locks=1*, *exclude_tags=None*, *style='auto'*, *with_rich=None*, *colors=1*, ***kwargs*)

Print info about the commands, optionally with rich

Example

```
>>> # xdoctest: +SKIP
>>> from cmd_queue.airflow_queue import *  # NOQA
>>> self = AirflowQueue()
>>> self.submit('date')
>>> self.print_commands()
>>> self.run()
```

rprint(*with_status=False*, *with_gaurds=False*, *with_locks=1*, *exclude_tags=None*, *style='auto'*, *with_rich=None*, *colors=1*, ***kwargs*)

Print info about the commands, optionally with rich

Example

```
>>> # xdoctest: +SKIP
>>> from cmd_queue.airflow_queue import *  # NOQA
>>> self = AirflowQueue()
>>> self.submit('date')
>>> self.print_commands()
>>> self.run()
```

cmd_queue.airflow_queue.**demo**()

Airflow requires initialization:
airflow db init

1.2.3 cmd_queue.base_queue module

exception cmd_queue.base_queue.**DuplicateJobError**

Bases: KeyError

exception cmd_queue.base_queue.**UnknownBackendError**

Bases: KeyError

class cmd_queue.base_queue.**Job**(*command=None*, *name=None*, *depends=None*, ***kwargs*)

Bases: NiceRepr

Base class for a job

class cmd_queue.base_queue.**Queue**

Bases: NiceRepr

Base class for a queue.

Use the `create` classmethod to make a concrete instance with an available backend.

change_backend(*backend*, ***kwargs*)

Create a new version of this queue with a different backend.

Currently metadata is not carried over. Submit an MR if you need this functionality.

Example

```
>>> from cmd_queue import Queue
>>> self = Queue.create(size=5, name='demo')
>>> self.submit('echo "Hello World"', name='job1a')
>>> self.submit('echo "Hello Revocable"', name='job1b')
>>> self.submit('echo "Hello Crushed"', depends=['job1a'], name='job2a')
>>> self.submit('echo "Hello Shadow"', depends=['job1b'], name='job2b')
>>> self.submit('echo "Hello Excavate"', depends=['job2a', 'job2b'], name='job3
↪')
>>> self.submit('echo "Hello Barrette"', depends=[], name='jobX')
>>> self.submit('echo "Hello Overwrite"', depends=['jobX'], name='jobY')
>>> self.submit('echo "Hello Giblet"', depends=['jobY'], name='jobZ')
>>> serial_backend = self.change_backend('serial')
>>> tmux_backend = self.change_backend('tmux')
>>> slurm_backend = self.change_backend('slurm')
>>> airflow_backend = self.change_backend('airflow')
>>> serial_backend.print_commands()
>>> tmux_backend.print_commands()
>>> slurm_backend.print_commands()
>>> airflow_backend.print_commands()
```

sync()

Mark that all future jobs will depend on the current sink jobs

Returns
a reference to the queue (for chaining)

Return type
Queue

`write()`
Writes the underlying files that defines the queue for whatever program will ingest it to run it.

`submit(command, **kwargs)`

Parameters

- **name** – specify the name of the job
- ****kwargs** – passed to `cmd_queue.serial_queue.BashJob`

`classmethod _backend_classes()`

`classmethod available_backends()`

`classmethod create(backend='serial', **kwargs)`
Main entry point to create a queue

Parameters
****kwargs** – environ (dict | None): environment variables name (str): queue name dpath (str): queue work directory gpus (int): number of gpus size (int): only for tmux queue, number of parallel queues

`write_network_text(reduced=True, rich='auto', vertical_chains=False)`

`print_commands(with_status=False, with_gaurds=False, with_locks=1, exclude_tags=None, style='colors', **kwargs)`

Parameters

- **with_status** (*bool*) – tmux / serial only, show bash status boilerplate
- **with_gaurds** (*bool*) – tmux / serial only, show bash guards boilerplate
- **with_locks** (*bool | int*) – tmux, show tmux lock boilerplate
- **exclude_tags** (*List[str] | None*) – if specified exclude jobs submitted with these tags.
- **style** (*str*) – can be 'colors', 'rich', or 'plain'
- ****kwargs** – extra backend-specific args passed to finalize_text

CommandLine

```
xdoctest -m cmd_queue.slurm_queue SlurmQueue.print_commands
xdoctest -m cmd_queue.serial_queue SerialQueue.print_commands
xdoctest -m cmd_queue.tmux_queue TMUXMultiQueue.print_commands
```

`rprint(**kwargs)`

`print_graph(reduced=True, vertical_chains=False)`
Renders the dependency graph to an "network text"

Parameters
reduced (*bool*) – if True only show the implicit dependency forest

_dependency_graph()

Builds a networkx dependency graph for the current jobs

Example

```
>>> from cmd_queue import Queue
>>> self = Queue.create(size=5, name='foo')
>>> job1a = self.submit('echo hello && sleep 0.5')
>>> job1b = self.submit('echo hello && sleep 0.5')
>>> job2a = self.submit('echo hello && sleep 0.5', depends=[job1a])
>>> job2b = self.submit('echo hello && sleep 0.5', depends=[job1b])
>>> job3 = self.submit('echo hello && sleep 0.5', depends=[job2a, job2b])
>>> jobX = self.submit('echo hello && sleep 0.5', depends=[])
>>> jobY = self.submit('echo hello && sleep 0.5', depends=[jobX])
>>> jobZ = self.submit('echo hello && sleep 0.5', depends=[jobY])
>>> graph = self._dependency_graph()
>>> self.print_graph()
```

monitor()

_coerce_style(*style='auto'*, *with_rich=None*, *colors=1*)

1.2.4 cmd_queue.cli_boilerplate module

This file defines a helper scriptconfig base config that can be used to help make cmd_queue CLIs so cmd_queue options are standardized and present at the top level.

CommandLine

```
xdoctest -m cmd_queue.cli_boilerplate __doc__:0
```

Example

```
>>> from cmd_queue.cli_boilerplate import CMDQueueConfig
>>> import scriptconfig as scfg
>>> import rich
>>> #
>>> class MyQueueCLI(CMDQueueConfig):
>>>     'A custom CLI that includes the cmd-queue boilerplate'
>>>     my_input_file = scfg.Value(None, help='some custom param')
>>>     my_num_steps = scfg.Value(3, help='some custom param')
>>>     is_small = scfg.Value(False, help='some custom param')
>>>     my_output_file = scfg.Value(None, help='some custom param')
>>> #
>>> def my_cli_main(cmdline=1, **kwargs):
>>>     config = MyQueueCLI.cli(cmdline=cmdline, data=kwargs)
>>>     rich.print('config = {}'.format(ub.urepr(config, nl=1)))
>>>     queue = config.create_queue()
>>>     #
```

(continues on next page)

(continued from previous page)

```
>>>     ###
>>>     # Custom code to submit jobs to the queue
>>>     #
>>>     job0 = queue.submit(f'echo "processing input file: {config.my_input_file}"',␣
↪name='ROOT-INPUT-JOB')
>>>     #
>>>     independent_outputs = []
>>>     for idx in range(config.my_num_steps):
>>>         job_t1 = queue.submit(f'echo "tree {idx}.S"', depends=[job0], name=f'jobname
↪{idx}.1')
>>>         if not config.is_small:
>>>             job_t2 = queue.submit(f'echo "tree {idx}.SL"', depends=[job_t1], name=f
↪'jobname{idx}.2')
>>>             job_t3 = queue.submit(f'echo "tree {idx}.SR"', depends=[job_t2], name=f
↪'jobname{idx}.3')
>>>             job_t4 = queue.submit(f'echo "tree {idx}.SRR"', depends=[job_t3], name=f
↪'jobname{idx}.4')
>>>             job_t5 = queue.submit(f'echo "tree {idx}.SRL"', depends=[job_t3], name=f
↪'jobname{idx}.5')
>>>             job_t6 = queue.submit(f'echo "tree {idx}.T"', depends=[job_t4, job_t5],␣
↪name=f'jobname{idx}.6')
>>>             job_t7 = queue.submit(f'echo "tree {idx}.SLT"', depends=[job_t2], name=f
↪'jobname{idx}.7')
>>>             independent_outputs.extend([job_t6, job_t2])
>>>         else:
>>>             independent_outputs.extend([job_t1])
>>>     #
>>>     queue.submit(f'echo "processing output file: {config.my_output_file}"',␣
↪depends=independent_outputs, name='FINAL-OUTPUT-JOB')
>>>     ###
>>>     #
>>>     config.run_queue(queue)
>>> #
>>> # Show what happens when you use the serial backend
>>> print('--------------------')
>>> print('--- DEMO SERIAL ---')
>>> print('--------------------')
>>> my_cli_main(
>>>     cmdline=0,
>>>     run=0,
>>>     print_queue=1,
>>>     print_commands=1,
>>>     backend='serial'
>>> )
>>> # Show what happens when you use the tmux backend
>>> print('------------------')
>>> print('--- DEMO TMUX ---')
>>> print('------------------')
>>> my_cli_main(
>>>     cmdline=0,
>>>     run=0,
>>>     print_queue=0,
```

(continues on next page)

(continued from previous page)

```
>>>     is_small=True,
>>>     my_num_steps=0,
>>>     print_commands=1,
>>>     backend='tmux'
>>> )
>>> # Show what happens when you use the slurm backend
>>> print('------------------')
>>> print('--- DEMO SLURM ---')
>>> print('------------------')
>>> my_cli_main(
>>>     cmdline=0,
>>>     run=0, backend='slurm',
>>>     print_commands=1,
>>>     print_queue=False,
>>>     slurm_options='''
>>>         partition: 'general-gpu'
>>>         account: 'default'
>>>         ntasks: 1
>>>         gres: 'gpu:1'
>>>         cpus_per_task: 4
>>>     '''
>>> )
>>> # xdoctest: +REQUIRES(--run)
>>> # Actually run with the defaults
>>> print('----------------')
>>> print('--- DEMO RUN ---')
>>> print('----------------')
>>> my_cli_main(cmdline=0, run=1, print_queue=0, print_commands=0)
```

class cmd_queue.cli_boilerplate.**CMDQueueConfig**(**args*, ***kwargs*)

Bases: DataConfig

A helper to carry around the common boilerplate for cmd-queue CLI's. The general usage is that you will inherit from this class and define config options your CLI cares about, however they must not overload any of the options specified here.

Usage will be to call *CMDQueueConfig.create_queue()* to initialize a queue based on these options, and then execute it with *CMDQueueConfig.run_queue()*. In this way you do not need to worry about this specific boilerplate when writing your application. See `cmd_queue.cli_boilerplate __doc__:0` for example usage.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

create_queue(***kwargs*)

Create an empty queue based on options specified in this config

Parameters

****kwargs** – extra args passed to cmd_queue.Queue.create

Returns

cmd_queue.Queue

run_queue(*queue*, *print_kwargs=None*, ***kwargs*)

Execute a queue with options based on this config.

Parameters

- **queue** (*cmd_queue.Queue*) – queue to run / report
- **print_kwargs** (*None | Dict*)

default = {'backend': <Value('tmux')>, 'other_session_handler': <Value('ask')>, 'print_commands': <Value('auto')>, 'print_queue': <Value('auto')>, 'queue_name': <Value(None)>, 'run': <Value(False)>, 'slurm_options': <Value(None)>, 'tmux_workers': <Value(8)>, 'virtualenv_cmd': <Value(None)>, 'with_textual': <Value('auto')>}

normalize()

1.2.5 cmd_queue.monitor_app module

class cmd_queue.monitor_app.**JobTable**(*table_fn=None*, ***kwargs*)

Bases: object

on_mount()

render()

class cmd_queue.monitor_app.**CmdQueueMonitorApp**(*table_fn*, *kill_fn=None*, ***kwargs*)

Bases: object

A Textual App to monitor jobs

classmethod demo()

This creates an app instance that we can run

CommandLine

```
xdoctest -m /home/joncrall/code/cmd_queue/cmd_queue/monitor_app.py␣
↪CmdQueueMonitorApp.demo:0 --interact
```

Example

```
>>> # xdoctest: +REQUIRES(module:textual)
>>> # xdoctest: +REQUIRES(--interact)
>>> from cmd_queue.monitor_app import CmdQueueMonitorApp
>>> self = CmdQueueMonitorApp.demo()
>>> self.run()
>>> print(f'self.graceful_exit={self.graceful_exit}')
```

async on_load(*event*) → None

async action_quit() → None

async on_mount(*event*) → None

1.2.6 cmd_queue.serial_queue module

References

https://jmmv.dev/2018/03/shell-readability-strict-mode.html https://stackoverflow.com/questions/13195655/bash-set-x-without-it-being-printed

cmd_queue.serial_queue.**indent**(*text*, *prefix=' '*)

Indents a block of text

Parameters

- **text** (*str*) – text to indent
- **prefix** (*str, default = ' '*) – prefix to add to each line

Returns

indented text

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> text = ['aaaa', 'bb', 'cc\n   dddd\n    ef\n']
>>> text = indent(text)
>>> print(text)
>>> text = indent(text)
>>> print(text)
```

Return type

str

class cmd_queue.serial_queue.**BashJob**(*command*, *name=None*, *depends=None*, *gpus=None*, *cpus=None*, *mem=None*, *bookkeeper=0*, *info_dpath=None*, *log=False*, *tags=None*, *allow_indent=True*, ***kwargs*)

Bases: *Job*

A job meant to run inside of a larger bash file. Analog of SlurmJob

Variables

- **name** (*str*) – a name for this job
- **pathid** (*str*) – a unique id based on the name and a hashed uuid
- **command** (*str*) – the shell command to run
- **depends** (*List[BashJob] | None*) – the jobs that this job depends on. This job will only run once all the dependencies have succesfully run.
- **bookkeeper** (*bool*) – flag indicating if this is a bookkeeping job or not
- **info_dpath** (*PathLike | None*) – where information about this job will be stored
- **log** (*bool*) – if True, output of the job will be tee-d and saved to a file, this can have interactions with normal stdout. Defaults to False.
- **tags** (*List[str] | str | None*) – a list of strings that can be used to group jobs or filter the queue or other custom purposes.
- **allow_indent** (*bool*) – In some cases indentation matters for the shell command. In that case ensure this is False at the cost of readability in the result script.

Todo:

- **[ ] What is a good name for a a list of jobs that must fail**
  for this job to run. Our current depends in analogous to slurm's afterok. What is a good variable name for afternotok? Do we wrap the job with some sort of negation, so we depend on the negation of the job?

CommandLine

```
xdoctest -m cmd_queue.serial_queue BashJob
```

Example

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> # Demo full boilerplate for a job with no dependencies
>>> self = BashJob('echo hi', 'myjob')
>>> self.print_commands(1, 1)
```

Example

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> # Demo full boilerplate for a job with dependencies
>>> dep = BashJob('echo hi', name='job1')
>>> conditionals = {'on_skip': ['echo "CUSTOM MESSAGE FOR WHEN WE SKIP A JOB"']}
>>> self = BashJob('echo hi', name='job2', depends=[dep])
>>> self.print_commands(1, 1, conditionals=conditionals)
```

finalize_text(*with_status=True*, *with_gaurds=True*, *conditionals=None*, ***kwargs*)

print_commands(*with_status=False*, *with_gaurds=False*, *with_rich=None*, *style='colors'*, ***kwargs*)

Print info about the commands, optionally with rich

Parameters

- **with_status** (*bool*) – tmux / serial only, show bash status boilerplate
- **with_gaurds** (*bool*) – tmux / serial only, show bash guards boilerplate
- **with_locks** (*bool*) – tmux, show tmux lock boilerplate
- **exclude_tags** (*List[str] | None*) – if specified exclude jobs submitted with these tags.
- **style** (*str*) – can be 'colors', 'rich', or 'plain'
- ****kwargs** – extra backend-specific args passed to finalize_text

CommandLine

```
xdoctest -m cmd_queue.serial_queue BashJob.print_commands
```

Example

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> self = SerialQueue('test-print-commands-serial-queue')
>>> self.submit('echo hi 1')
>>> self.submit('echo hi 2')
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=1, with_gaurds=1, style='rich')
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, with_gaurds=1, style='rich')
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, with_gaurds=0, style='rich')
```

class cmd_queue.serial_queue.**SerialQueue**(*name=''*, *dpath=None*, *rootid=None*, *environ=None*, *cwd=None*, ***kwargs*)

Bases: *Queue*

A linear job queue written to a single bash file

Example

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> self = SerialQueue('test-serial-queue', rootid='test-serial')
>>> job1 = self.submit('echo "this job fails" && false')
>>> job2 = self.submit('echo "this job works" && true')
>>> job3 = self.submit('echo "this job wont run" && true', depends=job1)
>>> self.print_commands(1, 1)
>>> self.run()
>>> state = self.read_state()
>>> print('state = {}'.format(ub.repr2(state, nl=1)))
```

Example

```
>>> # Test case where a job fails
>>> from cmd_queue.serial_queue import *  # NOQA
>>> self = SerialQueue('test-serial-queue', rootid='test-serial')
>>> job1 = self.submit('echo "job1 fails" && false')
>>> job2 = self.submit('echo "job2 never runs"', depends=[job1])
>>> job3 = self.submit('echo "job3 never runs"', depends=[job2])
>>> job4 = self.submit('echo "job4 passes" && true')
>>> job5 = self.submit('echo "job5 fails" && false', depends=[job4])
>>> job6 = self.submit('echo "job6 never runs"', depends=[job5])
>>> job7 = self.submit('echo "job7 never runs"', depends=[job4, job2])
>>> job8 = self.submit('echo "job8 never runs"', depends=[job4, job1])
>>> self.print_commands(1, 1)
```

(continues on next page)

(continued from previous page)

```
>>> self.run()
>>> self.read_state()
```

property pathid

A path-safe identifier for file names

classmethod is_available()

This queue is always available.

order_jobs()

Ensure jobs within a serial queue are topologically ordered. Attempts to preserve input ordering.

finalize_text(*with_status=True*, *with_gaurds=True*, *with_locks=True*, *exclude_tags=None*)

Create the bash script that will:

1. Run all of the jobs in this queue.
2. Track the results.
3. Prevent jobs with unmet dependencies from running.

add_header_command(*command*)

print_commands(**args*, ***kwargs*)

Print info about the commands, optionally with rich

CommandLine

```
xdoctest -m cmd_queue.serial_queue SerialQueue.print_commands
```

Example

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> self = SerialQueue('test-serial-queue')
>>> self.submit('echo hi 1')
>>> self.submit('echo hi 2')
>>> self.submit('echo boilerplate', tags='boilerplate')
>>> self.print_commands(with_status=True)
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, exclude_tags='boilerplate')
```

rprint(**args*, ***kwargs*)

Print info about the commands, optionally with rich

CommandLine

```
xdoctest -m cmd_queue.serial_queue SerialQueue.print_commands
```

Example

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> self = SerialQueue('test-serial-queue')
>>> self.submit('echo hi 1')
>>> self.submit('echo hi 2')
>>> self.submit('echo boilerplate', tags='boilerplate')
>>> self.print_commands(with_status=True)
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, exclude_tags='boilerplate')
```

run(*block=True*, *system=False*, *shell=1*, *capture=True*, *mode='bash'*, *verbose=3*, ***kw*)

job_details()

read_state()

cmd_queue.serial_queue.**_bash_json_dump**(*json_fmt_parts*, *fpath*)

Make a printf command that dumps a json file indicating some status in a bash environment.

Parameters

- **List[Tuple[str, str, str]]** – A list of 3-tupels indicating the name of the json key, the printf code, and the bash expression to fill the printf code.
- **fpath** (*str*) – where bash should write the json file

Returns

the bash that will perform the printf

Return type

str

Example

```
>>> from cmd_queue.serial_queue import _bash_json_dump
>>> json_fmt_parts = [
>>>     ('home', '%s', '$HOME'),
>>>     ('const', '%s', 'MY_CONSTANT'),
>>>     ('ps2', '"%s"', '$PS2'),
>>> ]
>>> fpath = 'out.json'
>>> dump_code = _bash_json_dump(json_fmt_parts, fpath)
>>> print(dump_code)
```

1.2.7 cmd_queue.slurm_queue module

Work in progress. The idea is to provide a TMUX queue and a SLURM queue that provide a common high level API, even though functionality might diverge, the core functionality of running processes asynchronously should be provided.

Notes

Installing and configuring SLURM See git@github.com:Erotemic/local.git init/setup_slurm.sh Or ~/local/init/setup_slurm.sh in my local checkout

SUBMIT COMMANDS WILL USE /bin/sh by default, not sure how to fix that properly. There are workarounds though.

CommandLine

```
xdoctest -m cmd_queue.slurm_queue __doc__
```

Example

```
>>> from cmd_queue.slurm_queue import *  # NOQA
>>> dpath = ub.Path.appdir('slurm_queue/tests')
>>> queue = SlurmQueue()
>>> job0 = queue.submit(f'echo "here we go"', name='root job')
>>> job1 = queue.submit(f'mkdir -p {dpath}', depends=[job0])
>>> job2 = queue.submit(f'echo "result=42" > {dpath}/test.txt ', depends=[job1])
>>> job3 = queue.submit(f'cat {dpath}/test.txt', depends=[job2])
>>> queue.print_commands()
>>> # xdoctest: +REQUIRES(--run)
>>> queue.run()
>>> # Can read the output of jobs after they are done.
>>> for job in queue.jobs:
>>>     print('-----------------')
>>>     print(f'job.name={job.name}')
>>>     if job.output_fpath.exists():
>>>         print(job.output_fpath.read_text())
>>>     else:
>>>         print('output does not exist')
```

cmd_queue.slurm_queue.**_coerce_mem**(*mem*)

Parameters

mem (*int | str*) – integer number of megabytes or a parseable string

Example

```
>>> from cmd_queue.slurm_queue import *  # NOQA
>>> print(_coerce_mem(30602))
>>> print(_coerce_mem('4GB'))
>>> print(_coerce_mem('32GB'))
>>> print(_coerce_mem('300000000 bytes'))
```

class cmd_queue.slurm_queue.**SlurmJob**(*command*, *name=None*, *output_fpath=None*, *depends=None*, *cpus=None*, *gpus=None*, *mem=None*, *begin=None*, *shell=None*, *tags=None*, ***kwargs*)

Bases: *Job*

Represents a slurm job that hasn't been submitted yet

Example

```
>>> from cmd_queue.slurm_queue import *  # NOQA
>>> self = SlurmJob('python -c print("hello world")', 'hi', cpus=5, gpus=1, mem=
↪'10GB')
>>> command = self._build_sbatch_args()
>>> print('command = {!r}'.format(command))
>>> self = SlurmJob('python -c print("hello world")', 'hi', cpus=5, gpus=1, mem=
↪'10GB', depends=[self])
>>> command = self._build_command()
>>> print(command)
```

_build_command(*jobname_to_varname=None*)

_build_sbatch_args(*jobname_to_varname=None*)

class cmd_queue.slurm_queue.**SlurmQueue**(*name=None*, *shell=None*, ***kwargs*)

Bases: *Queue*

CommandLine

```
xdoctest -m cmd_queue.slurm_queue SlurmQueue
```

Example

```
>>> from cmd_queue.slurm_queue import *  # NOQA
>>> self = SlurmQueue()
>>> job0 = self.submit('echo "hi from $SLURM_JOBID"', begin=0)
>>> job1 = self.submit('echo "hi from $SLURM_JOBID"', depends=[job0])
>>> job2 = self.submit('echo "hi from $SLURM_JOBID"', depends=[job1])
>>> job3 = self.submit('echo "hi from $SLURM_JOBID"', depends=[job2])
>>> job4 = self.submit('echo "hi from $SLURM_JOBID"', depends=[job3])
>>> job5 = self.submit('echo "hi from $SLURM_JOBID"', depends=[job4])
>>> job6 = self.submit('echo "hi from $SLURM_JOBID"', depends=[job0])
>>> job7 = self.submit('echo "hi from $SLURM_JOBID"', depends=[job5, job6])
```

(continues on next page)

(continued from previous page)

```
>>> self.write()
>>> self.print_commands()
>>> # xdoctest: +REQUIRES(--run)
>>> if not self.is_available():
>>>     self.run()
```

Example

```
>>> from cmd_queue.slurm_queue import *  # NOQA
>>> self = SlurmQueue(shell='/bin/bash')
>>> self.add_header_command('export FOO=bar')
>>> job0 = self.submit('echo "$FOO"')
>>> job1 = self.submit('echo "$FOO"', depends=job0)
>>> job2 = self.submit('echo "$FOO"')
>>> job3 = self.submit('echo "$FOO"', depends=job2)
>>> self.sync()
>>> job4 = self.submit('echo "$FOO"')
>>> self.sync()
>>> job5 = self.submit('echo "$FOO"')
>>> self.print_commands()
```

classmethod is_available()

Determines if we can run the slurm queue or not.

submit(*command*, ***kwargs*)

add_header_command(*command*)

order_jobs()

finalize_text(*exclude_tags=None*, ***kwargs*)

run(*block=True*, *system=False*, ***kw*)

monitor(*refresh_rate=0.4*)

Monitor progress until the jobs are done

kill()

read_state()

print_commands(**args*, ***kwargs*)

Print info about the commands, optionally with rich

Parameters

- **exclude_tags** (*List[str] | None*) – if specified exclude jobs submitted with these tags.
- **style** (*str*) – can be ‘colors’, ‘rich’, or ‘plain’

CommandLine

```
xdoctest -m cmd_queue.slurm_queue SlurmQueue.print_commands
```

Example

```
>>> from cmd_queue.slurm_queue import *  # NOQA
>>> self = SlurmQueue('test-slurm-queue')
>>> self.submit('echo hi 1')
>>> self.submit('echo hi 2')
>>> self.submit('echo boilerplate', tags='boilerplate')
>>> self.print_commands(with_status=True)
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, exclude_tags='boilerplate')
```

rprint(**args*, ***kwargs*)

Print info about the commands, optionally with rich

Parameters

- **exclude_tags** (*List[str] | None*) – if specified exclude jobs submitted with these tags.
- **style** (*str*) – can be ‘colors’, ‘rich’, or ‘plain’

CommandLine

```
xdoctest -m cmd_queue.slurm_queue SlurmQueue.print_commands
```

Example

```
>>> from cmd_queue.slurm_queue import *  # NOQA
>>> self = SlurmQueue('test-slurm-queue')
>>> self.submit('echo hi 1')
>>> self.submit('echo hi 2')
>>> self.submit('echo boilerplate', tags='boilerplate')
>>> self.print_commands(with_status=True)
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, exclude_tags='boilerplate')
```

1.2.8 cmd_queue.tmux_queue module

A very simple queue based on tmux and bash

It should be possible to add more functionality, such as:

- [x] A linear job queue - via one tmux shell
- [x] Mulitple linear job queues - via multiple tmux shells
- **[x] Ability to query status of jobs - tmux script writes status to a** file, secondary thread reads is.

- [x] Unique identifier per queue
- **[ ] Central scheduler - given that we can know when a job is done**
  a central scheduling process can run in the background, check the status of existing jobs, and spawn new jobs. — Maybe not needed.
- **[X] Dependencies between jobs - given a central scheduler, it can**
  only spawn a new job if a its dependencies have been met.
- **[ ] GPU resource requirements - if a job indicates how much of a**
  particular resources it needs, the scheduler can only schedule the next job if it "fits" given the resources taken by the current running jobs.
- **[x] Duck typed API that uses Slurm if available. Slurm is a robust**
  full featured queuing system. If it is available we should make it easy for the user to swap the tmux queue for slurm.
- **[x] Duck typed API that uses subprocesses. Tmux is not always available,**
  we could go even lighter weight and simply execute a subprocess that does the same thing as the linear queue. The downside is you don't get the nice tmux way of looking at the status of what the jobs are doing, but that doesn't matter in debugged automated workflows, and this does seem like a nice simple utility. Doesnt seem to exist elsewhere either, but my search terms might be wrong.
- [ ] Handle the case where some jobs need the GPU and others do not

Example

```
>>> import cmd_queue
>>> queue = cmd_queue.Queue.create(backend='tmux')
>>> job1 = queue.submit('echo "Hello World" && sleep 0.1')
>>> job2 = queue.submit('echo "Hello Kitty" && sleep 0.1', depends=[job1])
>>> if queue.is_available():
>>>     queue.run()
```

class cmd_queue.tmux_queue.**TMUXMultiQueue**(*size=1*, *name=None*, *dpath=None*, *rootid=None*, *environ=None*, *gpus=None*, *gres=None*)

Bases: *Queue*

Create multiple sets of jobs to start in detatched tmux sessions

CommandLine

```
xdoctest -m cmd_queue.tmux_queue TMUXMultiQueue:0
xdoctest -m cmd_queue.tmux_queue TMUXMultiQueue:2
```

Example

```
>>> from cmd_queue.serial_queue import *  # NOQA
>>> self = TMUXMultiQueue(1, 'test-tmux-queue')
>>> job1 = self.submit('echo hi 1 && false')
>>> job2 = self.submit('echo hi 2 && true')
>>> job3 = self.submit('echo hi 3 && true', depends=job1)
>>> self.print_commands()
>>> self.print_graph()
>>> if self.is_available():
>>>     self.run(block=True, onexit='capture', check_other_sessions=0)
```

Example

```
>>> from cmd_queue.tmux_queue import *  # NOQA
>>> import random
>>> rng = random.Random(54425367001)
>>> self = TMUXMultiQueue(1, 'real-world-usecase', gpus=[0, 1])
>>> def add_edge(name, depends):
>>>     if name is not None:
>>>         _depends = [self.named_jobs[n] for n in depends if n is not None]
>>>         self.submit(f'echo name={name}, depends={depends} && sleep 0.1',␣
↪name=name, depends=_depends)
>>> def add_branch(suffix):
>>>     f = 0.3
>>>     pred = f'pred{suffix}' if rng.random() > f else None
>>>     track = f'track{suffix}' if rng.random() > f else None
>>>     actclf = f'actclf{suffix}' if rng.random() > f else None
>>>     pxl_eval = f'pxl_eval{suffix}' if rng.random() > f else None
>>>     trk_eval = f'trk_eval{suffix}' if rng.random() > f else None
>>>     act_eval = f'act_eval{suffix}' if rng.random() > f else None
>>>     add_edge(pred, [])
>>>     add_edge(track, [pred])
>>>     add_edge(actclf, [pred])
>>>     add_edge(pxl_eval, [pred])
>>>     add_edge(trk_eval, [track])
>>>     add_edge(act_eval, [actclf])
>>> for i in range(3):
>>>     add_branch(str(i))
>>> self.print_commands()
>>> self.print_graph()
>>> if self.is_available():
>>>     self.run(block=1, onexit='', check_other_sessions=0)
```

Example

```
>>> from cmd_queue.tmux_queue import TMUXMultiQueue
>>> self = TMUXMultiQueue(size=2, name='foo')
>>> print('self = {!r}'.format(self))
>>> job1 = self.submit('echo hello && sleep 0.5')
>>> job2 = self.submit('echo world && sleep 0.5', depends=[job1])
>>> job3 = self.submit('echo foo && sleep 0.5')
>>> job4 = self.submit('echo bar && sleep 0.5')
>>> job5 = self.submit('echo spam && sleep 0.5', depends=[job1])
>>> job6 = self.submit('echo spam && sleep 0.5')
>>> job7 = self.submit('echo err && false')
>>> job8 = self.submit('echo spam && sleep 0.5')
>>> job9 = self.submit('echo eggs && sleep 0.5', depends=[job8])
>>> job10 = self.submit('echo bazbiz && sleep 0.5', depends=[job9])
>>> self.write()
>>> self.print_commands()
>>> if self.is_available():
>>>     self.run(check_other_sessions=0)
>>>     self.monitor()
>>>     self.current_output()
>>>     self.kill()
```

Example

```
>>> # Test complex failure case
>>> from cmd_queue import Queue
>>> self = Queue.create(size=2, name='demo-complex-failure', backend='tmux')
>>> # Submit a binary tree that fails at different levels
>>> for idx in range(2):
>>>     # Level 0
>>>     job1000 = self.submit('true')
>>>     # Level 1
>>>     job1100 = self.submit('true', depends=[job1000])
>>>     job1200 = self.submit('false', depends=[job1000], name=f'false0_{idx}')
>>>     # Level 2
>>>     job1110 = self.submit('true', depends=[job1100])
>>>     job1120 = self.submit('false', depends=[job1100], name=f'false1_{idx}')
>>>     job1210 = self.submit('true', depends=[job1200])
>>>     job1220 = self.submit('true', depends=[job1200])
>>>     # Level 3
>>>     job1111 = self.submit('true', depends=[job1110])
>>>     job1112 = self.submit('false', depends=[job1110], name=f'false2_{idx}')
>>>     job1121 = self.submit('true', depends=[job1120])
>>>     job1122 = self.submit('true', depends=[job1120])
>>>     job1211 = self.submit('true', depends=[job1210])
>>>     job1212 = self.submit('true', depends=[job1210])
>>>     job1221 = self.submit('true', depends=[job1220])
>>>     job1222 = self.submit('true', depends=[job1220])
>>> # Submit a chain that fails in the middle
>>> chain1 = self.submit('true', name='chain1')
```

(continues on next page)

(continued from previous page)

```
>>> chain2 = self.submit('true', depends=[chain1], name='chain2')
>>> chain3 = self.submit('false', depends=[chain2], name='chain3')
>>> chain4 = self.submit('true', depends=[chain3], name='chain4')
>>> chain5 = self.submit('true', depends=[chain4], name='chain5')
>>> # Submit 4 loose passing jobs
>>> for _ in range(4):
>>>     self.submit('true', name=f'loose_true{_}')
>>> # Submit 4 loose failing jobs
>>> for _ in range(4):
>>>     self.submit('false', name=f'loose_false{_}')
>>> self.print_commands()
>>> self.print_graph()
>>> if self.is_available():
>>>     self.run(with_textual=False, check_other_sessions=0)
```

classmethod is_available()

Determines if we can run the tmux queue or not.

_new_workers(*start=0*)

_semaphore_wait_command(*flag_fpaths*, *msg*)

TODO: use flock?

_semaphore_signal_command(*flag_fpath*)

order_jobs()

TODO: ability to shuffle jobs subject to graph constraints

Example

```
>>> from cmd_queue.tmux_queue import *  # NOQA
>>> self = TMUXMultiQueue(5, 'foo')
>>> job1a = self.submit('echo hello && sleep 0.5')
>>> job1b = self.submit('echo hello && sleep 0.5')
>>> job2a = self.submit('echo hello && sleep 0.5', depends=[job1a])
>>> job2b = self.submit('echo hello && sleep 0.5', depends=[job1b])
>>> job3 = self.submit('echo hello && sleep 0.5', depends=[job2a, job2b])
>>> self.print_commands()
```

self.run(block=True, check_other_sessions=0)

Example

```
>>> from cmd_queue.tmux_queue import *  # NOQA
>>> self = TMUXMultiQueue(5, 'foo')
>>> job0 = self.submit('true')
>>> job1 = self.submit('true')
>>> job2 = self.submit('true', depends=[job0])
>>> job3 = self.submit('true', depends=[job1])
>>> #job2c = self.submit('true', depends=[job1a, job1b])
>>> #self.sync()
```

(continues on next page)

(continued from previous page)

```
>>> job4 = self.submit('true', depends=[job2, job3, job1])
>>> job5 = self.submit('true', depends=[job4])
>>> job6 = self.submit('true', depends=[job4])
>>> job7 = self.submit('true', depends=[job4])
>>> job8 = self.submit('true', depends=[job5])
>>> job9 = self.submit('true', depends=[job6])
>>> job10 = self.submit('true', depends=[job6])
>>> job11 = self.submit('true', depends=[job7])
>>> job12 = self.submit('true', depends=[job10, job11])
>>> job13 = self.submit('true', depends=[job4])
>>> job14 = self.submit('true', depends=[job13])
>>> job15 = self.submit('true', depends=[job4])
>>> job16 = self.submit('true', depends=[job15, job13])
>>> job17 = self.submit('true', depends=[job4])
>>> job18 = self.submit('true', depends=[job17])
>>> job19 = self.submit('true', depends=[job14, job16, job17])
>>> self.print_graph(reduced=False)
...
Graph:
╾╾ foo-job-0
    └─ foo-job-2
        └─ foo-job-4  foo-job-3, foo-job-1
            ├─ foo-job-5
            │   └─ foo-job-8
            ├─ foo-job-6
            │   ├─ foo-job-9
            │   └─ foo-job-10
            │       └─ foo-job-12  foo-job-11
            ├─ foo-job-7
            │   └─ foo-job-11
            │       └─  ...
            ├─ foo-job-13
            │   ├─ foo-job-14
            │   │   └─ foo-job-19  foo-job-16, foo-job-17
            │   └─ foo-job-16  foo-job-15
            │       └─  ...
            ├─ foo-job-15
            │   └─  ...
            └─ foo-job-17
                ├─ foo-job-18
                └─  ...
╾╾ foo-job-1
    ├─ foo-job-3
    │   └─  ...
    └─  ...
>>> self.print_commands()
>>> # self.run(block=True)
```

Example

```
>>> from cmd_queue.tmux_queue import *  # NOQA
>>> self = TMUXMultiQueue(2, 'test-order-case')
>>> self.submit('echo slow1', name='slow1')
>>> self.submit('echo fast1', name='fast1')
>>> self.submit('echo slow2', name='slow2')
>>> self.submit('echo fast2', name='fast2')
>>> self.submit('echo slow3', name='slow3')
>>> self.submit('echo fast3', name='fast3')
>>> self.submit('echo slow4', name='slow4')
>>> self.submit('echo fast4', name='fast4')
>>> self.print_graph(reduced=False)
>>> self.print_commands()
```

add_header_command(*command*)

Adds a header command run at the start of each queue

finalize_text(***kwargs*)

write()

kill_other_queues(*ask_first=True*)

Find other tmux sessions that look like they were started with cmd_queue and kill them.

handle_other_sessions(*other_session_handler*)

run(*block=True*, *onfail='kill'*, *onexit=''*, *system=False*, *with_textual='auto'*, *check_other_sessions=None*, *other_session_handler='auto'*, ***kw*)

Execute the queue.

Parameters

other_session_handler (*str*) – How to handle potentially conflicting existing tmux runners with the same queue name. Can be ‘kill’, ‘ask’, or ‘ignore’, or ‘auto’ - which defaults to ‘ask’ if stdin is available and ‘kill’ if it is not.

read_state()

serial_run()

Hack to run everything without tmux. This really should be a different “queue” backend.

See Serial Queue instead

monitor(*refresh_rate=0.4*, *with_textual='auto'*)

Monitor progress until the jobs are done

CommandLine

```
xdoctest -m cmd_queue.tmux_queue TMUXMultiQueue.monitor:0
xdoctest -m cmd_queue.tmux_queue TMUXMultiQueue.monitor:1 --interact
```

Example

```
>>> # xdoctest: +REQUIRES(--interact)
>>> from cmd_queue.tmux_queue import *  # NOQA
>>> self = TMUXMultiQueue(size=3, name='test-queue-monitor')
>>> job = None
>>> for i in range(10):
>>>     job = self.submit('sleep 2 && echo "hello 2"', depends=job)
>>> job = None
>>> for i in range(10):
>>>     job = self.submit('sleep 3 && echo "hello 2"', depends=job)
>>> job = None
>>> for i in range(5):
>>>     job = self.submit('sleep 5 && echo "hello 2"', depends=job)
>>> self.print_commands()
>>> if self.is_available():
>>>     self.run(block=True, check_other_sessions=0)
```

Example

```
>>> # xdoctest: +REQUIRES(env:INTERACTIVE_TEST)
>>> from cmd_queue.tmux_queue import *  # NOQA
>>> # Setup a lot of longer running jobs
>>> n = 2
>>> self = TMUXMultiQueue(size=n, name='demo_cmd_queue')
>>> first_job = None
>>> for i in range(n):
...     prev_job = None
...     for j in range(4):
...         command = f'sleep 1 && echo "This is job {i}.{j}"'
...         job = self.submit(command, depends=prev_job)
...         prev_job = job
...         first_job = first_job or job
>>> command = f'sleep 1 && echo "this is the last job"'
>>> job = self.submit(command, depends=[prev_job, first_job])
>>> self.print_commands(style='rich')
>>> self.print_graph()
>>> if self.is_available():
...     self.run(block=True, other_session_handler='kill')
```

_textual_monitor()

_simple_rich_monitor(*refresh_rate=0.4*)

_build_status_table()

print_commands(**args*, ***kwargs*)

Print info about the commands, optionally with rich

Parameters

- **with_status** (*bool*) – tmux / serial only, show bash status boilerplate
- **with_gaurds** (*bool*) – tmux / serial only, show bash guards boilerplate

- **with_locks** (*bool*) – tmux, show tmux lock boilerplate
- **exclude_tags** (*List[str] | None*) – if specified exclude jobs submitted with these tags.
- **style** (*str*) – can be 'colors', 'rich', or 'plain'
- ****kwargs** – extra backend-specific args passed to finalize_text

CommandLine

```
xdoctest -m cmd_queue.tmux_queue TMUXMultiQueue.print_commands
```

Example

```
>>> from cmd_queue.tmux_queue import *  # NOQA
>>> self = TMUXMultiQueue(size=2, name='test-print-commands-tmux-queue')
>>> self.submit('echo hi 1', name='job1')
>>> self.submit('echo boilerplate job1', depends='job1', tags='boilerplate')
>>> self.submit('echo hi 2', log=False)
>>> self.submit('echo hi 3')
>>> self.submit('echo hi 4')
>>> self.submit('echo hi 5', log=False, name='job5')
>>> self.submit('echo boilerplate job2', depends='job5', tags='boilerplate')
>>> self.submit('echo hi 6', name='job6', depends='job5')
>>> self.submit('echo hi 7', name='job7', depends='job5')
>>> self.submit('echo boilerplate job3', depends=['job6', 'job7'], tags=
↪'boilerplate')
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=1, with_gaurds=1, with_locks=1, style='rich
↪')
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, with_gaurds=1, with_locks=1, style='rich
↪')
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, with_gaurds=0, with_locks=0, style='rich
↪')
>>> print('\n\n---\n\n')
>>> self.print_commands(with_status=0, with_gaurds=0, with_locks=0,
...             style='auto', exclude_tags='boilerplate')
```

current_output()

_print_commands()

_kill_commands()

capture()

kill()

_tmux_current_sessions()

cmd_queue.tmux_queue.**has_stdin**()

1.3 Module contents

The cmd_queue module is a tool that lets users define a DAG of bash commands. This DAG can be executed in a lightweight tmux backend, or a heavyweight slurm backend, or in simple serial mode that runs in the foreground thread. Rich provides monitoring / live control. For more information see the gitlab README. There is also a Google slides presentation that gives a high level overview.

The following examples show how to use the cmd_queue API in Python. For examples of the Bash API see: *cmd_queue.__main__*.

Example

```
>>> # The available backends classmethod lets you know which backends
>>> # your system has access to. The "serial" backend should always be
>>> # available. Everything else requires some degree of setup (tmux
>>> # is the easiest, just install it, no configuration needed).
>>> import cmd_queue
>>> print(cmd_queue.Queue.available_backends())  # xdoctest: +IGNORE_WANT
['serial', 'tmux', 'slurm']
```

Example

```
>>> # The API to submit jobs is the same regardless of the backend.
>>> # Job dependencies can be specified by name, or by the returned
>>> # job objects.
>>> import cmd_queue
>>> queue = cmd_queue.Queue.create(backend='serial')
>>> job1a = queue.submit('echo "Hello World" && sleep 0.1', name='job1a')
>>> job1b = queue.submit('echo "Hello Revocable" && sleep 0.1', name='job1b')
>>> job2a = queue.submit('echo "Hello Crushed" && sleep 0.1', depends=[job1a], name=
↪'job2a')
>>> job2b = queue.submit('echo "Hello Shadow" && sleep 0.1', depends=[job1b], name='job2b
↪')
>>> job3 = queue.submit('echo "Hello Excavate" && sleep 0.1', depends=[job2a, job2b],␣
↪name='job3')
>>> jobX = queue.submit('echo "Hello Barrette" && sleep 0.1', depends=[], name='jobX')
>>> jobY = queue.submit('echo "Hello Overwrite" && sleep 0.1', depends=[jobX], name='jobY
↪')
>>> jobZ = queue.submit('echo "Hello Giblet" && sleep 0.1', depends=[jobY], name='jobZ')
...
>>> # Use print_graph to get a "network text" representation of the DAG
>>> # This gives you a sense of what jobs can run in parallel
>>> queue.print_graph(reduced=False)
Graph:
╾╾ job1a
   └─ job2a
      └─ job3  job2b
╾╾ job1b
   └─ job2b
      └─  ...
╾╾ jobX
```

(continues on next page)

(continued from previous page)

```
    └─ jobY
        └─ jobZ
>>> # The purpose of command queue is not to run the code, but to
>>> # generate the code that would run the code.
>>> # The print_commands method gives you the gist of the code
>>> # command queue would run. Flags can be given to modify conciseness.
>>> queue.print_commands(style='plain')
# --- ...
#!/bin/bash
# Written by cmd_queue ...

# ----
# Jobs
# ----

#
### Command 1 / 8 - job1a
echo "Hello World" && sleep 0.1
#
### Command 2 / 8 - job1b
echo "Hello Revocable" && sleep 0.1
#
### Command 3 / 8 - job2a
echo "Hello Crushed" && sleep 0.1
#
### Command 4 / 8 - job2b
echo "Hello Shadow" && sleep 0.1
#
### Command 5 / 8 - job3
echo "Hello Excavate" && sleep 0.1
#
### Command 6 / 8 - jobX
echo "Hello Barrette" && sleep 0.1
#
### Command 7 / 8 - jobY
echo "Hello Overwrite" && sleep 0.1
#
### Command 8 / 8 - jobZ
echo "Hello Giblet" && sleep 0.1
>>> # Different backends have different ways of executing the
>>> # the underlying DAG, but it always boils down to: generate the code
>>> # that would execute your jobs.
>>> #
>>> # For the TMUX queue it boils down to writing a bash script for
>>> # sessions that can run in parallel, and a bash script that submits
>>> # them as different sessions (note: locks exist but are omitted here)
>>> tmux_queue = queue.change_backend('tmux', size=2)
>>> tmux_queue.print_commands(style='plain', with_locks=0)
# --- ...sh
#!/bin/bash
# Written by cmd_queue ...
# ----
```

(continues on next page)

(continued from previous page)

```
# Jobs
# ----
#
### Command 1 / 3 - jobX
echo "Hello Barrette" && sleep 0.1
#
### Command 2 / 3 - jobY
echo "Hello Overwrite" && sleep 0.1
#
### Command 3 / 3 - jobZ
echo "Hello Giblet" && sleep 0.1
# --- ...sh
#!/bin/bash
# Written by cmd_queue ...
# ----
# Jobs
# ----
#
### Command 1 / 4 - job1a
echo "Hello World" && sleep 0.1
#
### Command 2 / 4 - job2a
echo "Hello Crushed" && sleep 0.1
#
### Command 3 / 4 - job1b
echo "Hello Revocable" && sleep 0.1
#
### Command 4 / 4 - job2b
echo "Hello Shadow" && sleep 0.1
# --- ...sh
#!/bin/bash
# Written by cmd_queue ...
# ----
# Jobs
# ----
#
### Command 1 / 1 - job3
echo "Hello Excavate" && sleep 0.1
# --- ...sh
#!/bin/bash
# Driver script to start the tmux-queue
echo "Submitting 8 jobs to a tmux queue"
### Run Queue: cmdq_unnamed_000_... with 3 jobs
tmux new-session -d -s cmdq_unnamed_000_... "bash"
tmux send -t cmdq_unnamed_... \
    "source ...sh" \
    Enter
### Run Queue: cmdq_unnamed_001_... with 4 jobs
tmux new-session -d -s cmdq_unnamed_001_... "bash"
tmux send -t cmdq_unnamed_001_... \
    "source ...sh" \
    Enter
```

(continues on next page)

(continued from previous page)

```
### Run Queue: cmdq_unnamed_002_... with 1 jobs
tmux new-session -d -s cmdq_unnamed_002_... "bash"
tmux send -t cmdq_unnamed_... \
    "source ...sh" \
    Enter
echo "Spread jobs across 3 tmux workers"
>>> # The slurm queue is very simple, it just constructs one bash file that is the
>>> # sbatch commands to submit your jobs. All of the other details are taken care of
>>> # by slurm itself.
>>> # xdoctest: +IGNORE_WANT
>>> slurm_queue = queue.change_backend('slurm')
>>> slurm_queue.print_commands(style='plain')
# --- ...sh
mkdir -p ".../logs"
JOB_000=$(sbatch --job-name="job1a" --output="/.../logs/job1a.sh" --wrap 'echo "Hello␣
↪World" && sleep 0.1' --parsable)
JOB_001=$(sbatch --job-name="job1b" --output="/.../logs/job1b.sh" --wrap 'echo "Hello␣
↪Revocable" && sleep 0.1' --parsable)
JOB_002=$(sbatch --job-name="jobX" --output="/.../logs/jobX.sh" --wrap 'echo "Hello␣
↪Barrette" && sleep 0.1' --parsable)
JOB_003=$(sbatch --job-name="job2a" --output="/.../logs/job2a.sh" --wrap 'echo "Hello␣
↪Crushed" && sleep 0.1' "--dependency=afterok:${JOB_000}" --parsable)
JOB_004=$(sbatch --job-name="job2b" --output="/.../logs/job2b.sh" --wrap 'echo "Hello␣
↪Shadow" && sleep 0.1' "--dependency=afterok:${JOB_001}" --parsable)
JOB_005=$(sbatch --job-name="jobY" --output="/.../logs/jobY.sh" --wrap 'echo "Hello␣
↪Overwrite" && sleep 0.1' "--dependency=afterok:${JOB_002}" --parsable)
JOB_006=$(sbatch --job-name="job3" --output="/.../logs/job3.sh" --wrap 'echo "Hello␣
↪Excavate" && sleep 0.1' "--dependency=afterok:${JOB_003}:${JOB_004}" --parsable)
JOB_007=$(sbatch --job-name="jobZ" --output="/.../logs/jobZ.sh" --wrap 'echo "Hello␣
↪Giblet" && sleep 0.1' "--dependency=afterok:${JOB_005}" --parsable)
>>> # The airflow backend is slightly different because it defines
>>> # DAGs with Python files, so we write a Python file instead of
>>> # a bash file. NOTE: the process of actually executing the airflow
>>> # DAG has not been finalized yet. (Help wanted)
>>> airflow_queue = queue.change_backend('airflow')
>>> airflow_queue.print_commands(style='plain')
# --- ...py
from airflow import DAG
from datetime import timezone
from datetime import datetime as datetime_cls
from airflow.operators.bash import BashOperator
now = datetime_cls.utcnow().replace(tzinfo=timezone.utc)
dag = DAG(
    'SQ',
    start_date=now,
    catchup=False,
    tags=['example'],
)
jobs = dict()
jobs['job1a'] = BashOperator(task_id='job1a', bash_command='echo "Hello World" && sleep␣
↪0.1', dag=dag)
jobs['job1b'] = BashOperator(task_id='job1b', bash_command='echo "Hello Revocable" &&␣
```

(continues on next page)

(continued from previous page)

```
↪sleep 0.1', dag=dag)
jobs['job2a'] = BashOperator(task_id='job2a', bash_command='echo "Hello Crushed" &&␣
↪sleep 0.1', dag=dag)
jobs['job2b'] = BashOperator(task_id='job2b', bash_command='echo "Hello Shadow" && sleep␣
↪0.1', dag=dag)
jobs['job3'] = BashOperator(task_id='job3', bash_command='echo "Hello Excavate" && sleep␣
↪0.1', dag=dag)
jobs['jobX'] = BashOperator(task_id='jobX', bash_command='echo "Hello Barrette" && sleep␣
↪0.1', dag=dag)
jobs['jobY'] = BashOperator(task_id='jobY', bash_command='echo "Hello Overwrite" &&␣
↪sleep 0.1', dag=dag)
jobs['jobZ'] = BashOperator(task_id='jobZ', bash_command='echo "Hello Giblet" && sleep 0.
↪1', dag=dag)
jobs['job2a'].set_upstream(jobs['job1a'])
jobs['job2b'].set_upstream(jobs['job1b'])
jobs['job3'].set_upstream(jobs['job2a'])
jobs['job3'].set_upstream(jobs['job2b'])
jobs['jobY'].set_upstream(jobs['jobX'])
jobs['jobZ'].set_upstream(jobs['jobY'])
```

Example

```
>>> # Given a Queue object, the "run" method will attempt to execute it
>>> # for you and give you a sense of progress.
>>> # xdoctest: +IGNORE_WANT
>>> import cmd_queue
>>> queue = cmd_queue.Queue.create(backend='serial')
>>> job1a = queue.submit('echo "Hello World" && sleep 0.1', name='job1a')
>>> job1b = queue.submit('echo "Hello Revocable" && sleep 0.1', name='job1b')
>>> job2a = queue.submit('echo "Hello Crushed" && sleep 0.1', depends=[job1a], name=
↪'job2a')
>>> job2b = queue.submit('echo "Hello Shadow" && sleep 0.1', depends=[job1b], name='job2b
↪')
>>> job3 = queue.submit('echo "Hello Excavate" && sleep 0.1', depends=[job2a, job2b],␣
↪name='job3')
>>> jobX = queue.submit('echo "Hello Barrette" && sleep 0.1', depends=[], name='jobX')
>>> jobY = queue.submit('echo "Hello Overwrite" && sleep 0.1', depends=[jobX], name='jobY
↪')
>>> jobZ = queue.submit('echo "Hello Giblet" && sleep 0.1', depends=[jobY], name='jobZ')
>>> # Using the serial queue simply executes all of the commands in order in
>>> # the current session. This behavior can be useful as a fallback or
>>> # for debugging.
>>> # Note: xdoctest doesnt seem to capture the set -x parts. Not sure why.
>>> queue.run(block=True, system=True)  # xdoctest: +IGNORE_WANT
─── START CMD ───
[ubelt.cmd] ...@...:...$ bash ...sh
+ echo 'Hello World'
Hello World
+ sleep 0.1
+ echo 'Hello Revocable'
```

(continues on next page)

(continued from previous page)

```
Hello Revocable
+ sleep 0.1
+ echo 'Hello Crushed'
Hello Crushed
+ sleep 0.1
+ echo 'Hello Shadow'
Hello Shadow
+ sleep 0.1
+ echo 'Hello Excavate'
Hello Excavate
+ sleep 0.1
+ echo 'Hello Barrette'
Hello Barrette
+ sleep 0.1
+ echo 'Hello Overwrite'
Hello Overwrite
+ sleep 0.1
+ echo 'Hello Giblet'
Hello Giblet
+ sleep 0.1
Command Queue Final Status:
{"status": "done", "passed": 8, "failed": 0, "skipped": 0, "total": 8, "name": "",
↪"rootid": "..."}
└─── END CMD ───
>>> # The TMUX queue does not show output directly by default (although
>>> # it does have access to methods that let it grab logs from tmux)
>>> # But normally you can attach to the tmux sessions to look at them
>>> # The default monitor will depend on if you have textual installed or not.
>>> # Another default behavior is that it will ask if you want to kill
>>> # previous command queue tmux sessions, but this can be disabled.
>>> import ubelt as ub
>>> if 'tmux' in cmd_queue.Queue.available_backends():
>>>     tmux_queue = queue.change_backend('tmux', size=2)
>>>     tmux_queue.run(with_textual='auto', check_other_sessions=False)
[ubelt.cmd] joncrall@calculex:~/code/cmd_queue$ bash /home/joncrall/.cache/cmd_queue/
↪tmux/unnamed_2022-07-27_cbfeedda/run_queues_unnamed.sh
submitting 8 jobs
jobs submitted

 tmux session name  status  passed  failed  skipped  total

┌ cmdq_unnamed_000 │ done   │ 3      │ 0      │ 0       │ 3     ┐
│ cmdq_unnamed_001 │ done   │ 4      │ 0      │ 0       │ 4     │
│ cmdq_unnamed_002 │ done   │ 1      │ 0      │ 0       │ 1     │
└ agg              │ done   │ 8      │ 0      │ 0       │ 8     ┘
>>> # The monitoring for the slurm queue is basic, and the extent to
>>> # which features can be added will depend on your slurm config.
>>> # Any other slurm monitoring tools can be used. There are plans
>>> # to implement a textual monitor based on the slurm logfiles.
>>> if 'slurm' in cmd_queue.Queue.available_backends():
>>>     slurm_queue = queue.change_backend('slurm')
```

(continues on next page)

(continued from previous page)

```
>>>     slurm_queue.run()
─── START CMD ───
[ubelt.cmd] ...sh
└─── END CMD ───
                          slurm-monitor

 num_running  num_in_queue  total_monitored  num_at_start

┌ 0           │ 31           │ 118              │ 118          │
└─────────────────────────────────────────────────────────────┘
>>> # xdoctest: +SKIP
>>> # Running airflow queues is not implemented yet
>>> if 'airflow' in cmd_queue.Queue.available_backends():
>>>     airflow_queue = queue.change_backend('airflow')
>>>     airflow_queue.run()
```

class cmd_queue.**Queue**

Bases: NiceRepr

Base class for a queue.

Use the `create` classmethod to make a concrete instance with an available backend.

change_backend(*backend*, ***kwargs*)

Create a new version of this queue with a different backend.

Currently metadata is not carried over. Submit an MR if you need this functionality.

Example

```
>>> from cmd_queue import Queue
>>> self = Queue.create(size=5, name='demo')
>>> self.submit('echo "Hello World"', name='job1a')
>>> self.submit('echo "Hello Revocable"', name='job1b')
>>> self.submit('echo "Hello Crushed"', depends=['job1a'], name='job2a')
>>> self.submit('echo "Hello Shadow"', depends=['job1b'], name='job2b')
>>> self.submit('echo "Hello Excavate"', depends=['job2a', 'job2b'], name='job3
↪')
>>> self.submit('echo "Hello Barrette"', depends=[], name='jobX')
>>> self.submit('echo "Hello Overwrite"', depends=['jobX'], name='jobY')
>>> self.submit('echo "Hello Giblet"', depends=['jobY'], name='jobZ')
>>> serial_backend = self.change_backend('serial')
>>> tmux_backend = self.change_backend('tmux')
>>> slurm_backend = self.change_backend('slurm')
>>> airflow_backend = self.change_backend('airflow')
>>> serial_backend.print_commands()
>>> tmux_backend.print_commands()
>>> slurm_backend.print_commands()
>>> airflow_backend.print_commands()
```

sync()

Mark that all future jobs will depend on the current sink jobs

Returns
a reference to the queue (for chaining)

Return type
Queue

`write()`
Writes the underlying files that defines the queue for whatever program will ingest it to run it.

`submit(command, **kwargs)`

Parameters

- **name** – specify the name of the job
- ****kwargs** – passed to `cmd_queue.serial_queue.BashJob`

`classmethod _backend_classes()`

`classmethod available_backends()`

`classmethod create(backend='serial', **kwargs)`
Main entry point to create a queue

Parameters
****kwargs** – environ (dict | None): environment variables name (str): queue name dpath (str): queue work directory gpus (int): number of gpus size (int): only for tmux queue, number of parallel queues

`write_network_text(reduced=True, rich='auto', vertical_chains=False)`

`print_commands(with_status=False, with_gaurds=False, with_locks=1, exclude_tags=None, style='colors', **kwargs)`

Parameters

- **with_status** (*bool*) – tmux / serial only, show bash status boilerplate
- **with_gaurds** (*bool*) – tmux / serial only, show bash guards boilerplate
- **with_locks** (*bool | int*) – tmux, show tmux lock boilerplate
- **exclude_tags** (*List[str] | None*) – if specified exclude jobs submitted with these tags.
- **style** (*str*) – can be 'colors', 'rich', or 'plain'
- ****kwargs** – extra backend-specific args passed to finalize_text

CommandLine

```
xdoctest -m cmd_queue.slurm_queue SlurmQueue.print_commands
xdoctest -m cmd_queue.serial_queue SerialQueue.print_commands
xdoctest -m cmd_queue.tmux_queue TMUXMultiQueue.print_commands
```

`rprint(**kwargs)`

`print_graph(reduced=True, vertical_chains=False)`
Renders the dependency graph to an "network text"

Parameters
reduced (*bool*) – if True only show the implicit dependency forest

_dependency_graph()

Builds a networkx dependency graph for the current jobs

Example

```
>>> from cmd_queue import Queue
>>> self = Queue.create(size=5, name='foo')
>>> job1a = self.submit('echo hello && sleep 0.5')
>>> job1b = self.submit('echo hello && sleep 0.5')
>>> job2a = self.submit('echo hello && sleep 0.5', depends=[job1a])
>>> job2b = self.submit('echo hello && sleep 0.5', depends=[job1b])
>>> job3 = self.submit('echo hello && sleep 0.5', depends=[job2a, job2b])
>>> jobX = self.submit('echo hello && sleep 0.5', depends=[])
>>> jobY = self.submit('echo hello && sleep 0.5', depends=[jobX])
>>> jobZ = self.submit('echo hello && sleep 0.5', depends=[jobY])
>>> graph = self._dependency_graph()
>>> self.print_graph()
```

monitor()

_coerce_style(*style='auto'*, *with_rich=None*, *colors=1*)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[SO56937691] https://stackoverflow.com/questions/56937691/making-yaml-ruamel-yaml-always-dump-lists-inline

PYTHON MODULE INDEX

C

cmd_queue, 41
cmd_queue.__init__, 1
cmd_queue.__main__, 16
cmd_queue.airflow_queue, 16
cmd_queue.base_queue, 18
cmd_queue.cli_boilerplate, 20
cmd_queue.monitor_app, 23
cmd_queue.serial_queue, 24
cmd_queue.slurm_queue, 29
cmd_queue.tmux_queue, 32
cmd_queue.util, 16
cmd_queue.util.richer, 9
cmd_queue.util.texter, 9
cmd_queue.util.textual_extensions, 9
cmd_queue.util.util_algo, 11
cmd_queue.util.util_networkx, 12
cmd_queue.util.util_tags, 12
cmd_queue.util.util_tmux, 12
cmd_queue.util.util_yaml, 13

INDEX

Symbols

_YamlRepresenter (*class in cmd_queue.util.util_yaml*), 13
_backend_classes() (*cmd_queue.Queue class method*), 48
_backend_classes() (*cmd_queue.base_queue.Queue class method*), 19
_bash_json_dump() (*in module cmd_queue.serial_queue*), 28
_build_command() (*cmd_queue.slurm_queue.SlurmJob method*), 30
_build_sbatch_args() (*cmd_queue.slurm_queue.SlurmJob method*), 30
_build_status_table() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 39
_capture_pane_command() (*cmd_queue.util.util_tmux.tmux static method*), 12
_coerce_mem() (*in module cmd_queue.slurm_queue*), 29
_coerce_style() (*cmd_queue.Queue method*), 49
_coerce_style() (*cmd_queue.base_queue.Queue method*), 20
_custom_pyaml_dumper() (*in module cmd_queue.util.util_yaml*), 13
_custom_ruaml_dumper() (*in module cmd_queue.util.util_yaml*), 13
_custom_ruaml_loader() (*in module cmd_queue.util.util_yaml*), 13
_dependency_graph() (*cmd_queue.Queue method*), 48
_dependency_graph() (*cmd_queue.base_queue.Queue method*), 19
_kill_commands() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 40
_kill_session_command() (*cmd_queue.util.util_tmux.tmux static method*), 12
_new_workers() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 36
_print_commands() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 40
_run_as_cls() (*cmd_queue.util.textual_extensions.InstanceRunnableApp class method*), 11
_run_as_instance() (*cmd_queue.util.textual_extensions.InstanceRunnab method*), 11
_semaphore_signal_command() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 36
_semaphore_wait_command() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 36
_simple_rich_monitor() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 39
_textual_monitor() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 39
_tmux_current_sessions() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 40

A

action_quit() (*cmd_queue.monitor_app.CmdQueueMonitorApp method*), 23
add_header_command() (*cmd_queue.serial_queue.SerialQueue method*), 27
add_header_command() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
add_header_command() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38
AirflowJob (*class in cmd_queue.airflow_queue*), 16
AirflowQueue (*class in cmd_queue.airflow_queue*), 16
available_backends() (*cmd_queue.base_queue.Queue class method*), 19
available_backends() (*cmd_queue.Queue class method*), 48

B

balanced_number_partitioning() (*in module*

cmd_queue.util.util_algo), 11
BashJob (*class in cmd_queue.serial_queue*), 24

C

capture() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 40
capture_pane() (*cmd_queue.util.util_tmux.tmux static method*), 12
change_backend() (*cmd_queue.base_queue.Queue method*), 18
change_backend() (*cmd_queue.Queue method*), 47
class_or_instancemethod (*class in cmd_queue.util.textual_extensions*), 9
cmd_queue
    module, 41
cmd_queue.__init__
    module, 1
cmd_queue.__main__
    module, 16
cmd_queue.airflow_queue
    module, 16
cmd_queue.base_queue
    module, 18
cmd_queue.cli_boilerplate
    module, 20
cmd_queue.monitor_app
    module, 23
cmd_queue.serial_queue
    module, 24
cmd_queue.slurm_queue
    module, 29
cmd_queue.tmux_queue
    module, 32
cmd_queue.util
    module, 16
cmd_queue.util.richer
    module, 9
cmd_queue.util.texter
    module, 9
cmd_queue.util.textual_extensions
    module, 9
cmd_queue.util.util_algo
    module, 11
cmd_queue.util.util_networkx
    module, 12
cmd_queue.util.util_tags
    module, 12
cmd_queue.util.util_tmux
    module, 12
cmd_queue.util.util_yaml
    module, 13
CMDQueueConfig (*class in cmd_queue.cli_boilerplate*), 22
CmdQueueMonitorApp (*class in cmd_queue.monitor_app*), 23
CodeBlock() (*cmd_queue.util.util_yaml.Yaml static method*), 16
coerce() (*cmd_queue.util.util_tags.Tags class method*), 12
coerce() (*cmd_queue.util.util_yaml.Yaml static method*), 14
create() (*cmd_queue.base_queue.Queue class method*), 19
create() (*cmd_queue.Queue class method*), 48
create_queue() (*cmd_queue.cli_boilerplate.CMDQueueConfig method*), 22
current_output() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 40

D

default (*cmd_queue.cli_boilerplate.CMDQueueConfig attribute*), 23
demo() (*cmd_queue.monitor_app.CmdQueueMonitorApp class method*), 23
demo() (*in module cmd_queue.airflow_queue*), 17
Dict() (*cmd_queue.util.util_yaml.Yaml static method*), 16
dumps() (*cmd_queue.util.util_yaml.Yaml static method*), 13
DuplicateJobError, 18

F

finalize_text() (*cmd_queue.airflow_queue.AirflowJob method*), 16
finalize_text() (*cmd_queue.airflow_queue.AirflowQueue method*), 17
finalize_text() (*cmd_queue.serial_queue.BashJob method*), 25
finalize_text() (*cmd_queue.serial_queue.SerialQueue method*), 27
finalize_text() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
finalize_text() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38

H

handle_other_sessions() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38
has_stdin() (*in module cmd_queue.tmux_queue*), 40

I

indent() (*in module cmd_queue.serial_queue*), 24
InlineList() (*cmd_queue.util.util_yaml.Yaml static method*), 16
InstanceRunnableApp (*class in cmd_queue.util.textual_extensions*), 10

intersection() (*cmd_queue.util.util_tags.Tags method*), 12
is_available() (*cmd_queue.airflow_queue.AirflowQueue class method*), 17
is_available() (*cmd_queue.serial_queue.SerialQueue class method*), 27
is_available() (*cmd_queue.slurm_queue.SlurmQueue class method*), 31
is_available() (*cmd_queue.tmux_queue.TMUXMultiQueue class method*), 36
is_topological_order() (*in module cmd_queue.util.util_networkx*), 12

J

Job (*class in cmd_queue.base_queue*), 18
job_details() (*cmd_queue.serial_queue.SerialQueue method*), 28
JobTable (*class in cmd_queue.monitor_app*), 23

K

kill() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
kill() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 40
kill_other_queues() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38
kill_session() (*cmd_queue.util.util_tmux.tmux static method*), 12

L

list_sessions() (*cmd_queue.util.util_tmux.tmux static method*), 12
load() (*cmd_queue.util.util_yaml.Yaml static method*), 13
loads() (*cmd_queue.util.util_yaml.Yaml static method*), 14

M

module
    cmd_queue, 41
    cmd_queue.__init__, 1
    cmd_queue.__main__, 16
    cmd_queue.airflow_queue, 16
    cmd_queue.base_queue, 18
    cmd_queue.cli_boilerplate, 20
    cmd_queue.monitor_app, 23
    cmd_queue.serial_queue, 24
    cmd_queue.slurm_queue, 29
    cmd_queue.tmux_queue, 32
    cmd_queue.util, 16
    cmd_queue.util.richer, 9
    cmd_queue.util.texter, 9
    cmd_queue.util.textual_extensions, 9
    cmd_queue.util.util_algo, 11
    cmd_queue.util.util_networkx, 12
    cmd_queue.util.util_tags, 12
    cmd_queue.util.util_tmux, 12
    cmd_queue.util.util_yaml, 13
monitor() (*cmd_queue.base_queue.Queue method*), 20
monitor() (*cmd_queue.Queue method*), 49
monitor() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
monitor() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38

N

normalize() (*cmd_queue.cli_boilerplate.CMDQueueConfig method*), 23

O

on_load() (*cmd_queue.monitor_app.CmdQueueMonitorApp method*), 23
on_mount() (*cmd_queue.monitor_app.CmdQueueMonitorApp method*), 23
on_mount() (*cmd_queue.monitor_app.JobTable method*), 23
order_jobs() (*cmd_queue.serial_queue.SerialQueue method*), 27
order_jobs() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
order_jobs() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 36

P

pathid (*cmd_queue.serial_queue.SerialQueue property*), 27
print_commands() (*cmd_queue.airflow_queue.AirflowQueue method*), 17
print_commands() (*cmd_queue.base_queue.Queue method*), 19
print_commands() (*cmd_queue.Queue method*), 48
print_commands() (*cmd_queue.serial_queue.BashJob method*), 25
print_commands() (*cmd_queue.serial_queue.SerialQueue method*), 27
print_commands() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
print_commands() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 39
print_graph() (*cmd_queue.base_queue.Queue method*), 19
print_graph() (*cmd_queue.Queue method*), 48

Q

Queue (*class in cmd_queue*), 47

Queue (*class in cmd_queue.base_queue*), 18

R

read_state() (*cmd_queue.serial_queue.SerialQueue method*), 28
read_state() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
read_state() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38
render() (*cmd_queue.monitor_app.JobTable method*), 23
rprint() (*cmd_queue.airflow_queue.AirflowQueue method*), 17
rprint() (*cmd_queue.base_queue.Queue method*), 19
rprint() (*cmd_queue.Queue method*), 48
rprint() (*cmd_queue.serial_queue.SerialQueue method*), 27
rprint() (*cmd_queue.slurm_queue.SlurmQueue method*), 32
run() (*cmd_queue.airflow_queue.AirflowQueue method*), 17
run() (*cmd_queue.serial_queue.SerialQueue method*), 28
run() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
run() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38
run() (*cmd_queue.util.textual_extensions.InstanceRunnableApp class method*), 11
run_queue() (*cmd_queue.cli_boilerplate.CMDQueueConfig method*), 22

S

serial_run() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38
SerialQueue (*class in cmd_queue.serial_queue*), 26
SlurmJob (*class in cmd_queue.slurm_queue*), 30
SlurmQueue (*class in cmd_queue.slurm_queue*), 30
str_presenter() (*cmd_queue.util.util_yaml._YamlRepresenter static method*), 13
submit() (*cmd_queue.airflow_queue.AirflowQueue method*), 17
submit() (*cmd_queue.base_queue.Queue method*), 19
submit() (*cmd_queue.Queue method*), 48
submit() (*cmd_queue.slurm_queue.SlurmQueue method*), 31
sync() (*cmd_queue.base_queue.Queue method*), 18
sync() (*cmd_queue.Queue method*), 47

T

Tags (*class in cmd_queue.util.util_tags*), 12
tmux (*class in cmd_queue.util.util_tmux*), 12
TMUXMultiQueue (*class in cmd_queue.tmux_queue*), 33

U

UnknownBackendError, 18

W

write() (*cmd_queue.base_queue.Queue method*), 19
write() (*cmd_queue.Queue method*), 48
write() (*cmd_queue.tmux_queue.TMUXMultiQueue method*), 38
write_network_text() (*cmd_queue.base_queue.Queue method*), 19
write_network_text() (*cmd_queue.Queue method*), 48

Y

Yaml (*class in cmd_queue.util.util_yaml*), 13