dequindre Documentation

Release 0.10.0

Nick Vogt

Mar 09, 2019

Contents

CHAPTER 1

Dequindre Is Easy to Setup

Anywhere Python goes, Dequindre can follow. Dequindre is written in pure python and is OS independent. All you need is to `pip install dequindre`.

CHAPTER 2

Dequindre Is Easy to Run

Dequindre makes it easy to run virtual environments. Dequindre supports virtualenv, pipenv, and conda environments.

CHAPTER 3

Dequindre Is Easy to Learn

You can run your first Dequindre workflow in minutes. Dequindre is less than 1000 lines of Python and fully documented. In contrast, Airflow v1.10.2 has 444 pages of docs.

CHAPTER 4

Your First Dequindre Schedule

Install dequindre from PyPI `pip install dequindre`. Then in the REPL,

```
>>> from dequindre import Task, DAG, Dequindre

>>> ## define tasks and environments
>>> boil_water = Task('./boil_water.py')
>>> steep_tea = Task('./steep_tea.py')
>>> drink_tea = Task('./drink_tea.py')

>>> ## define runtime dependencies
>>> make_tea = DAG(dependencies={
...     steep_tea: boil_water,
...     drink_tea: steep_tea
... })

>>> ## create schedules
>>> dq = Dequindre(make_tea)
>>> dq.get_schedules()
defaultdict(<class 'set'>, {
    1: {Task(./boil_water.py)},
    2: {Task(./steep_tea.py)},
    3: {Task(./drink_tea.py)}})

>>> ## run tasks if the files exist.
>>> dq.run_tasks()
Running Task(./boil_water.py)

I am boiling water...

Running Task(./steep_tea.py)

I am steeping tea...

Running Task(./drink_tea.py)
```

(continues on next page)

(continued from previous page)

```
I am drinking tea...
```

You can run the tasks by copy-pasting the following python code into the commented files.

```
# pour_water.py
print("I'm pouring water...")
```

```
# boil_water.py
print("I'm boiling water...")
```

```
# steep_tea.py
print("I'm steeping tea...")
```

CHAPTER 5

Features

- **Automated workflow scheduling**
- **Pure Python**: Relies entirely on Python built-ins to reduce bugs and complexity
- **Cross-Python compatible**: Supports Python 2 and Python 3
- **Cross-platform**: Windows and Unix style OS environments
- **Run your Python tasks in any pre-defined environments**
  - dequindre facilitates **virtualenv**, **conda**, and **pipenv** environments
- **Supports dynamic workflow configuration** also seen in Airflow
- **Documented** examples and configuration

CHAPTER 6

Extras

6.1 License

This project is licensed under the MIT License - see the LICENSE file for details.

6.2 Versioning

We use SemVer for versioning. For the versions available, see the tags on this repository.

6.3 Contribute

If you're interested in contributing to Dequindre, raise an issue, make a pull request to *dev*, and reach out to the author, vogt4nick.

Please read our contribution guidelines for details on our code of conduct, and the process for submitting pull requests to us.

6.4 Acknowledgements

Thank you, Dynatrace, for facilitating the early development of Dequindre during Innovation Day, February 2019.

6.5 User Guide

6.5.1 API Documentation

Dequindre Module

Task, DAG, and Dequindre classes

This module defines the Task, DAG, and Dequindre classes. Tasks are intended to hold task-level data. DAGs are intended to hold relationships between Tasks. Dequindre schedules Tasks in accordance with the DAG.

class dequindre.**DAG**(*, *tasks: set = None*, *dependencies: dict = None*)
Bases: object

Define a DAG and relationships between tasks.

A DAG is a directed acyclic graph with tasks and dependencies as nodes and directed edges respectively. You have the option to define all the tasks and dependencies at once if you prefer that syntax.

Note: Not obviously, a DAG may contain more than one graph. Also not obviously, new Tasks defined by edges are automatically added to the set of tasks.

---

tasks
set of *Task* – The set of all tasks. Dequindre will try to run every task in this attribute.

_edges (`dict` of `Task`
set of *Task*): A dict of directed edges from one Task to a set of Tasks. Access directly at your own peril.

add_dependencies(*d: Dict[dequindre.Task, Set[dequindre.Task]]*) → None
Add multiple dependencies to DAG

Parameters **d (‘dict‘ of ‘Task‘** – *set* of *Task*): An adjacency dict mapping downstream Tasks to possibly many upstream tasks.

Note: If any tasks do not yet exist in DAG, the task will automatically be added to the dag.

Examples

```
>>> from dequindre import Task, DAG
>>> boil_water = Task('boil_water.py')
>>> prep_infuser = Task('prep_infuser.py')
>>> steep_tea = Task('steep_tea.py')
>>> dag = DAG()
>>> dag.add_dependencies({steep_tea: {boil_water, prep_infuser}})
```

add_dependency(*task: dequindre.Task*, *depends_on: dequindre.Task*) → None
Add dependency to DAG.

Parameters

- **task** (*Task*) – The downstream task.
- **depends_on** (*Task*) – The upstream task.

Note: If either task does not yet exist in DAG, the task will automatically be added to the dag.

Examples

```
>>> from dequindre import Task, DAG
>>> boil_water = Task('boil_water.py')
>>> steep_tea = Task('steep_tea.py')
>>> dag = DAG()
>>> dag.add_dependency(steep_tea, depends_on=boil_water)
```

add_task(*task: dequindre.Task*) → None
Add a task to the set of tasks

Parameters **task** (*Task*) – A Task object.

add_tasks(*tasks: set*) → None
Add multiple tasks to the set of tasks.

Parameters **tasks** (*set* of *Task*) – Tasks to be added to the DAG.

get_downstream() → dict
Return adjacency dict of downstream Tasks.

Returns *set* of *Task*

Return type *dict* of *Task*

get_sinks() → set
Return the set of sink Tasks (Tasks with no downstream dependencies)

Returns *set* of *Task*

get_sources() → set
Return the set of source Tasks (Tasks with no upstream dependencies)

Returns *set* of *Task*

get_upstream() → dict
Return adjacency dict of upstream Tasks

Returns *set* of *Task*

Return type *dict* of *Task*

is_cyclic() → bool
Detect if the DAG is cyclic.

Returns True if cycle detected. False otherwise.

remove_task(*task: dequindre.Task*) → None
Remove task from the set of tasks and remove any related edges

Parameters **task** (*Task*) – A task to be removed from the DAG.

remove_tasks(*tasks: set*) → None
Remove multiple tasks from the set of tasks and any related edges

Parameters **tasks** (*set* of *Task*) – Tasks to be removed from the DAG.

class dequindre.**Dequindre**(*dag: dequindre.DAG*)
Bases: object

The Dequindre scheduler handles all the scheduling computations.

dag
DAG – A copy of the originally supplied DAG. This attribute is trimmed while planning the schedule.

original_dag
DAG – The originally supplied DAG. Used to refresh dag after the schedule is planned.

get_schedules() → Dict[int, Set[dequindre.Task]]
Schedule tasks by priority level.

Returns *set* of *Task*

Return type *dict* of *int*

For example, make_tea -> pour_tea -> drink_tea will give the dict

{1: {make_tea}, 2: {pour_tea}, 3: {drink_tea}}

get_task_schedules() → Dict[dequindre.Task, int]
Define schedule priority level for each task

Returns *int*

Return type *dict* of *Task*

Example

make_tea -> pour_tea -> drink_tea will give the dict {

make_tea: 1, pour_tea: 2, drink_tea: 3

}

refresh_dag() → None
Create a deepcopy of the original_dag.

run_task(*task: dequindre.Task*) → None
Run the python file defined by Task.loc in the environment defined by the Task.env

Parameters **task** (*Task*) – The task to be run.

run_tasks(*error_handling: str = 'soft'*) → None
Run all tasks on the DAG.

Parameters **error_handling** (*str*) – Either 'soft' or 'hard'. 'hard' error handling will abort the schedule after the first error.

class dequindre.**Task**(*loc: str*, *env: str = 'python'*)
Bases: object

Define a Task and its relevant attributes.

Note: Tasks with the same loc and env are equal.

---

loc
str – location of the python script that runs the task.

env
str, optional – Which environment to run.

Commons Submodule

Shortcuts to defining tasks and environments.

It's not uncommon for tasks to share a similar parent directory or structure. Likewise, environments almost always share the same parent directory or structure. The commons module makes well-founded assumptions about these structures to improve readability.

dequindre.commons.**common_conda_env**(*common_prefix: str*, *common_suffix: str = None*) → Callable

Quickly construct a path to a common conda environment

conda follows the structure: */path/to/conda/envs/{{my_env}}/bin/python*

Parameters

- **common_prefix** (*str*) – The file path before the environment name.
- **common_suffix** (*str, optional*) – The file path after the environment name.

Returns Function to shorten env specification

Example

```
>>>
>>> from dequindre.commons import common_conda_env
>>> with common_conda_env('/path/to/conda/envs') as env:
...     python27 = env('python27')
...     python36 = env('python36')
...
>>> python27
'/path/to/conda/envs/python27/bin/python'
```

dequindre.commons.**common_pipenv**(*common_prefix: str = '.'*, *common_suffix: str = None*) → Callable

Quickly construct a path to a common pipenv environment

pipenv follows the structure: */path/to/{{my_env}}/Scripts/python*

Parameters

- **common_prefix** (*str*) – The file path before the environment name.
- **common_suffix** (*str, optional*) – The file path after the environment name.

Returns Function to shorten env specification

Example

```
>>>
>>> from dequindre.commons import common_pipenv
>>> with common_pipenv('/path/to/tea-envs') as env:
...     python27 = env('python27')
...     python36 = env('python36')
...
>>> python27
'/path/to/tea-envs/python27/Scripts/python'
```

dequindre.commons.**common_task**(*loc_pattern: str*, *common_env: str = 'python'*)
Create tasks with a common parent path and environment.

Lots of tasks will use the same environment or same directory. These *commons* reduce duplicate code.

Parameters

- **loc_pattern** (*str*) – {}-formatted parent path.
- **common_env** (*str, optional*) – environment.

Example

```
>>> from dequindre.commons import common_task
>>> with common_task('./tea-tasks/{}/main.py') as T:
...     boil_water = T('boil_water')
...     steep_tea  = T('steep_tea')
...     drink_tea  = T('drink_tea')
>>> boil_water
Task(./tea-tasks/boil_water/main.py)
```

dequindre.commons.**common_venv**(*common_prefix: str = '.'*, *common_suffix: str = None*) → Callable
Quickly construct a path to a common virtualenv environment

venv follows the structure: */path/to/{{my_env}}/Scripts/python*

Parameters

- **common_prefix** (*str*) – The file path before the environment name.
- **common_suffix** (*str, optional*) – The file path after the environment name.

Returns Function to shorten env specification

Example

```
>>>
>>> from dequindre.commons import common_venv
>>> with common_venv('./tea-envs') as env:
...     python27 = env('python27')
...     python36 = env('python36')
...
>>> python27
'./tea-envs/python27/Scripts/python'
```

Exceptions Submodule

Exceptions that are unique to dequindre.

exception dequindre.exceptions.**CyclicGraphError**
Bases: Exception

Dequindre will generate an infinite schedule given a cyclic graph

exception dequindre.exceptions.**EarlyAbortError**
Bases: Exception

Used in conjunction with Dequindre.run_tasks() for error handling

6.5.2 Cookbook

Example files

This cookbook will make use of three sample python files.

```
## ./boil_water.py
print("I am boiling water...")
```

```
## ./steep_tea.py
print("I am steeping tea...")
```

```
## ./pour_tea.py
print("I am pouring tea...")
```

We also use Git Bash as the terminal. Bash commands work on windows and unix machines unless otherwise stated.

Tasks

Configure a Task

```
>>> from dequindre import Task
>>> pour_tea = Task('./pour_tea.py')
>>> pour_tea
Task(./pour_tea.py)
>>> pour_tea.loc
'./pour_tea.py'
>>> pour_tea.env
'python'
```

Note that that the python environment defaulted to 'python'. To use different environments, we'll need to define them first.

Commons

The `commons` submodule removes clutter from your config files. Even small workflows share many parent directories and environments. To this end, the `commons` submodule makes heavy use of context managers for readability.

Common Tasks

```
>>> from dequindre.commons import common_task

>>> common_prefix = '/long/path/to/tea-tasks'
>>> with common_task(common_prefix) as T:
...     pour_tea = T(loc='pour_tea.py')
...     drink_tea = T(loc='drink_tea.py')
...
>>> pour_tea
Task(/long/path/to/tea-tasks/pour_tea.py)
>>> drink_tea
Task(/long/path/to/tea-tasks/drink_tea.py)
```

(continues on next page)

(continued from previous page)

```
>>> pour_tea.env
'python'
```

The resulting task definitions are much easier to read.

Common Environments

Virtual environments get ugly fast, and they're best kept out of sight for many users.

```
>>> from dequindre import Task
>>> from dequindre.commons import common_venv

>>> common_prefix = '/my/very/long/path'
>>> with common_venv(common_prefix) as E:
...     tea_env = E('tea-env')
...     biscuit_env = E('biscuit-env')
...
>>> tea_env
'/my/very/long/path/tea-env/bin/python'
>>> drink_tea = Task('./drink_tea.py', tea_env)
>>> drink_tea.env
'/my/very/long/path/tea-env/bin/python'
```

Notice that `common_venv` filled in the expected suffix: `/bin/python`. You can override this behavior with the `common_suffix` argument.

The same functionality is also supported for pipenv environments and conda environments through the `common_pipenv` and `common_conda_env` functions respectively.

DAGs

Configure a DAG

```
>>> from dequindre import Task, DAG

>>> ## define tasks
>>> boil_water = Task('./boil_water.py')
>>> steep_tea = Task('./steep_tea.py')
>>> pour_tea = Task('./pour_tea.py')

>>> make_tea = DAG()
>>> make_tea.add_dependencies({
...       steep_tea: boil_water,
...       pour_tea: steep_tea
...   })
```

Dequindre Schedulers

The Dequindre scheduler is the last major object in dequindre. After defining your tasks and task dependencies in the DAG, you can create a Dequindre scheduler.

```
>>> from dequindre import Task, DAG, Dequindre

>>> ## define tasks
>>> boil_water = Task('./boil_water.py')
>>> steep_tea = Task('./steep_tea.py')
>>> pour_tea = Task('./pour_tea.py')

>>> make_tea = DAG()
>>> make_tea.add_dependencies({
...       steep_tea: boil_water,
...       pour_tea: steep_tea
...   })

>>> dq = Dequindre(make_tea)
>>> dq.get_schedules()
defaultdict(<class 'set'>, {
    1: {Task(boil_water.py)},
    2: {Task(steep_tea.py)},
    3: {Task(pour_tea.py)}})
>>> dq.run_tasks()

Running Task(./boil_water.py)

I am boiling water...

Running Task(./steep_tea.py)

I am steeping tea...

Running Task(./pour_tea.py)

I am pouring tea...
```

Error Handling

By default, Dequindre uses soft error handling; if one task fails, Dequindre assumes it's a non-critical error and continues on. But we don't always want this behavior. Instead, if one task fails, we want the whole schedule to fail.

`Dequindre.run_tasks()` has an optional `error_handling` method that takes one of two values: `error_handling='soft'` or `error_handling='hard'`. The latter will raise an `EarlyAbortError` if any of the tasks fail.

Bringing It All Together

After reviewing the cook book so far, we're ready to write an optimized Dequindre schedule. This works as a good reference when you're building your first schedules.

```
from pprint import pprint

from dequindre import Task, DAG, Dequindre
from dequindre.commons import common_task, common_conda_env


def run_schedule():
```

(continues on next page)

(continued from previous page)

```
    print('Starting run-my-schedule...')

    CONDA_PREFIX = '/opt/conda/envs'
    with common_conda_env(CONDA_PREFIX) as conda_env:
        python27 = conda_env('python27')
        python36 = conda_env('python36')

    TASK_PATTERN = '/opt/my-tasks/{}/main.py'
    with common_task(TASK_PATTERN, python27) as T:
        leave_home    = T('leave_home')
        get_fuel      = T('get_fuel')
        get_groceries = T('get_groceries')

    with common_task(TASK_PATTERN, python36) as T:
        pay_rent      = T('pay_rent')
        return_home   = T('return_home')
        make_dinner   = T('make_dinner')
        go_to_bed     = T('go_to_bed')

    dag = DAG(tasks={
        leave_home, get_fuel, get_groceries,
        pay_rent, return_home, make_dinner, go_to_bed
    })
    dag.add_dependencies({
        get_fuel: leave_home,
        get_groceries: leave_home,
        pay_rent: leave_home,
        return_home: {get_fuel, get_groceries, pay_rent},
        make_dinner: {return_home, get_groceries},
        go_to_bed: make_dinner
    })

    dq = Dequindre(dag)
    schedules = dq.get_schedules()
    pprint(schedules)
    # {1: {Task(/opt/my-tasks/leave_home/main.py)},
    #  2: {Task(/opt/my-tasks/get_fuel/main.py),
    #      Task(/opt/my-tasks/pay_rent/main.py),
    #      Task(/opt/my-tasks/get_groceries/main.py)},
    #  3: {Task(/opt/my-tasks/return_home/main.py)},
    #  4: {Task(/opt/my-tasks/make_dinner/main.py)},
    #  5: {Task(/opt/my-tasks/go_to_bed/main.py)}}

    dq.run_tasks()


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

6.5.3 Philosophy

Dequindre is built for amateurs and professionals with the most essential features in mind. It also functions as a learning tool for those without the time or resources to setup the requisite architecture for a full-featured scheduler.

Inspired by the complexity of Airflow, Dequindre leans heavilty on three lines of the Zen of Python:

1. Explicit is better than implicit
2. Simple is better than complex
3. Readability counts

From this starting point, Dequindre offers

- Minimal abstraction
- No Python dependencies; no third-party bugs
- Single-container deployment
- Legible source code; fewer than 1000 lines
- Fast, dynamic workflow configuration

Python Module Index

d

Index