Alyeska

Release 0.3.0a0

Oct 08, 2019

Contents

CHAPTER 1

API Documentation

1.1 Compose Module

Task, DAG, and Composer classes

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

class alyeska.compose.**Composer**(*dag: alyeska.compose.DAG*)

Bases: object

The Composer handles all the scheduling computations.

dag

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

Type *DAG*

original_dag

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

Type *DAG*

classmethod from_yaml(*p: pathlib.Path*) → alyeska.compose.Composer

Create a Composer from a compose.yaml file

Returns A Composer representation of compose.yaml

Return type *Composer*

get_schedules() → Dict[int, Set[alyeska.compose.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[alyeska.compose.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.

class alyeska.compose.**DAG**(*, *tasks: set = {}*, *upstream_dependencies: dict = {}*, *downstream_dependencies: dict = {}*)
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
The set of all tasks. Composer will try to run every task in this attribute.

Type *set* of *Task*

_edges (`dict` of `Task`
set of *Task*): Maps tasks to their downstream dependencies. Access directly at your own peril. e.g. A -> B -> C, not C -> B -> A

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

Parameters **(dict of Task** (*d*) – *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 Composer 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: alyeska.compose.Task*, *depends_on: alyeska.compose.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 Composer 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: alyeska.compose.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.

classmethod from_yaml (*p: pathlib.Path*) → alyeska.compose.DAG
Create a DAG from a compose.yaml file

Returns Directed Acyclic Graph representation of compose.yaml

Return type *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: alyeska.compose.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.

static validate_dependency (*d*)

class alyeska.compose.**Task** (*loc: pathlib.Path*, *env: str = '/home/docs/checkouts/readthedocs.org/user_builds/alyeska/envs/latest/bin/python'*, *validate_loc: bool = False*)
Bases: object

Define a Task and its relevant attributes.

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

---

loc
location of the python script that runs the task.

Type pathlib.Path

env
Which environment to run.

Type str, optional

env

loc

1.2 Locksmith Module

Fetch credentials from AWS Secrets Manager.

Usage:

```
>>> import boto3
>>> import alyeska.locksmith as ls
>>> session = boto3.Session()  # fetch creds from .aws/credentials
>>> secret_name = "my-super-secret-secret"
>>> secret = ls.get_secret(session, secret_name)
```

alyeska.locksmith.**get_secret** (*session: boto3.session.Session*, *secret_name: str*, *region_name: str = 'us-east-1'*) → dict
Get secret from secretsmanager using an established session.

See boto3.amazonaws.com/v1/documentation/api/latest/guide/secrets-manager.html

Parameters

- **session** (*boto3.Session*) – [description]
- **secret_name** (*str*) – [description]

- **region_name** (*str, optional*) – [description]. Defaults to "us-east-1".

Returns Secret as dict

Return type dict

alyeska.locksmith.**mfa_from_str**(*json_str: str*, *, *include_expiration=False*) → dict

Create credentials dict from credentials as a json string.

This function is a thin wrapper around json.loads

Parameters

- **json_str** (*str*) – String containing a json object. e.g. '" {

  "Credentials": { "AccessKeyId": "FAKEACCESSKEY", "SecretAccessKey": "Fake+Secret9Access-Key", "SessionToken": "f4k3-SE5510N_t0k3n", "Expiration": "2019-07-30T00:14:27Z"

  }

- **include_expiration** (*bool, optional*) – Whether to include expiration in returned json. Defaults to False.

Returns

with types as

{ "aws_access_key_id": str, "aws_secret_access_key": str, "aws_session_token": str, "expiration": datetime,

}

Return type dict

Example

```
>>> import dynatrace_locksmith as ls
>>> creds = ls.mfa_from_str(json_str)
>>> creds
{
    "Credentials": {
        "AccessKeyId": "1234567890",
        "SecretAccessKey": "qwertyuiop",
        "SessionToken": "asdfghjklzxcvbnm",
        "Expiration": "2018-11-02T05:15:21Z"
    }
}
```

```
>>> session = boto3.Session(**creds)
```

1.3 Logging Module

alyeska.logging submodule for configuring basic logs.

alyeska.logging.**config_logging**(***kwargs*)

Default logger configuration

e.g. 2019-08-13 12:01:14.334 UTC | INFO | This is a message

alyeska.logging.**log_scope_change**(*func*)
Log when programs enter and exit the decorated function

alyeska.logging.**sample**()

1.4 Red Pandas Module

alyeska redpandas module for smoother pandas/redshift functionality

alyeska.redpandas.**assert_table_exists**(*cnxn: psycopg2.extensions.connection*, *schema: str*, *table: str*) → None
Check that the table actually exists

Parameters

- **cnxn** (*psycopg2.extensions.connection*) – [description]
- **schema** (*str*) – [description]
- **table** (*str*) – [description]

Raises MissingTableError – If the target schema.table does not exist

alyeska.redpandas.**generate_insert_queries**(*curs: psycopg2.extensions.cursor*, *insert_table: str*, *df: pandas.core.frame.DataFrame*, *, *chunksize: int = 10000*) → Coroutine[T_co, T_contra, V_co]
Generator that helps insert_pandas_into. Assumes totally valid arguments, and colnames must match the schema of the insert table.

Parameters

- **curs** (*psycopg2.extensions.cursor*) – Connection used to insert to table
- **insert_table** (*str*) – Target table in database
- **df** (*pd.DataFrame*) – Pandas dataframe that will be inserted
- **chunksize** (*int, optional*) – How many rows to write per insert. Defaults to 10000.

Returns None

alyeska.redpandas.**insert_pandas_into**(*cnxn: psycopg2.extensions.connection*, *insert_table: str*, *df: pandas.core.frame.DataFrame*, *, *chunksize: int = 10000*) → None
Open connection and insert df into insert_table.

Parameters

- **cnxn** (*psycopg2.extensions.connection*) – Connection used to insert to table
- **insert_table** (*str*) – Target table in database
- **df** (*pd.DataFrame*) – Pandas dataframe that will be inserted
- **chunksize** (*int, optional*) – How many rows to write per insert. Defaults to 10000.

Returns [description]

Return type None

1.5 SQL Agent Module

Similar functionality to SQL Server Agent. Flush with functionality to automate SQL tasks.

alyeska.sqlagent.**execute_sql**(*cnxn: psycopg2.extensions.connection*, *cmd: str*) → None
Open *cnxn* and pass the *cmd* argument.

? Should we handle EOF errors?

Parameters

- **cnxn** (*psycopg2.extensions.connection*) – Connection used to execute command.
- **cmd** (*str*) – SQL command to be executed.

Returns None

alyeska.sqlagent.**execute_tasks**(*cnxn: psycopg2.extensions.connection*, **tasks*) → None
Execute the SQL in each task argument in order

Parameters **cnxn** (*psycopg2.extensions.connection*) – [description]

alyeska.sqlagent.**find_sql_files**(*sql_dir: pathlib.Path*, *include_subdirs: bool = True*) → Coroutine[pathlib.Path, None, pathlib.Path]
Find SQL files in the given directory.

Parameters

- **sql_dir** (*pathlib.Path*) – The directory to look for SQL files
- **include_subdirs** (*bool*) – Whether to include subdirectories in the search

Returns [description]

Return type Coroutine[pathlib.Path, None, pathlib.Path]

alyeska.sqlagent.**gather_subtasks**(*d: Dict[KT, VT]*) → collections.OrderedDict
Declare subtasks and log messages in order

Parameters **d** (*Dict*) – a dict-like object mapping subtasks to log messages

Returns map tasks to log messages in order of execution

Return type OrderedDict

alyeska.sqlagent.**plan_tasks**(*sql_dir: pathlib.Path*) → List[pathlib.Path]
Generate an ordered sequence of SQL files.

Parameters **sql_dir** (*pathlib.Path*) – Where to look for SQL files.

Returns An ordered sequence of filepaths.

Return type List[pathlib.Path]

Notes

plan_tasks doesn't return a generator here because the sorting step creates a list. Returning this sorted list as a generator would just create computational overhead.

alyeska.sqlagent.**process_batch**(*cnxn: psycopg2.extensions.connection*, *sql_dir: pathlib.Path*) → None
Find SQL files in sql_dir and execute as batch process

Parameters

- **cnxn** (*psycopg2.extensions.connection*) – [description]
- **sql_dir** (*str*) – [description]

Returns [description]

Return type None

alyeska.sqlagent.**run_sql**(*cnxn: psycopg2.extensions.connection*, *fp: pathlib.Path*, *msg: str*) → None

Run SQL: Read from file and execute with connection.

Parameters

- **cnxn** (*psycopg2.extensions.connection*) – Connection used to execute the SQL
- **fp** (*pathlib.Path*) – Filepath where target SQL is stored (relative or absolute)
- **msg** (*str*) – Message for logger when running subtask

Returns None

alyeska.sqlagent.**run_subtasks**(*cnxn: psycopg2.extensions.connection, subtasks: Dict[pathlib.Path, str]*) → None

Fetch SQL files and run them in order.

Parameters

- **cnxn** (*psycopg2.extensions.connection*) – Database connection used to run subtasks.
- **subtasks** (*OrderedDict[pathlib.Path, str]*) – OrderedDict containing paths to sql files mapped to the text read by logger.

Returns None

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

S

T

V