d6tflow Documentation

Release 0.1

nn

Jul 20, 2023

Contents

For data scientists and data engineers, d6tflow is a python library which makes it easier to build data workflows.

CHAPTER 1

Installation

Follow github instructions https://github.com/d6t/d6tflow#installation

CHAPTER 2

Benefits of using d6tflow

See 4 Reasons Why Your Machine Learning Code is Probably Bad

CHAPTER 3

Quickstart

See https://github.com/d6t/d6tflow/blob/master/docs/example-ml.md

CHAPTER 4

Real-life project template

https://github.com/d6t/d6tflow-template

CHAPTER 5

Transition to d6tflow from typical scripts

[5 Step Guide to Scalable Deep Learning Pipelines with d6tflow](https://htmlpreview.github.io/?https://github.com/d6t/d6t-python/blob/master/blogs/blog-20190813-d6tflow-pytorch.html)

CHAPTER 6

Parameter Management

Intelligent parameter management is one of the most powerful features of d6tflow. New users often have questions on parameter management, this is an important section to read.

CHAPTER 7

User Guide

7.1 Quickstart

See https://github.com/d6t/d6tflow/blob/master/docs/example-ml.md

7.2 Transition to d6tflow

7.2.1 Current Workflow Using Functions

Your code currently probably looks like the example below. How do you turn it into a d6tflow workflow?

```
import pandas as pd

def get_data():
    data = pd.read_csv('rawdata.csv')
    data = clean(data)
    data.to_pickle('data.pkl')

def preprocess(data):
    data = scale(data)
    return data

# execute workflow
get_data()
df_train = pd.read_pickle('data.pkl')
do_preprocess = True
if do_preprocess:
    df_train = preprocess(df_train)
```

7.2.2 Workflow Using d6tflow Tasks

In a d6tflow workflow, you define your own task classes and then execute the workflow by running the final downstream task which will automatically run required upstream dependencies.

The function-based workflow example will transform to this:

```
import d6tflow
import pandas as pd

class TaskGetData(d6tflow.tasks.TaskPqPandas):

    # no dependency

    def run(): # from `def get_data()`
        data = pd.read_csv('rawdata.csv')
        data = clean(data)
        self.save(data) # save output data

class TaskProcess(d6tflow.tasks.TaskPqPandas):
    do_preprocess = luigi.BoolParameter(default=True) # optional parameter

    def requires(self):
        return TaskGetData() # define dependency

    def run(self):
        data = self.input().load() # load input data
        if self.do_preprocess:
            data = scale(data) # # from `def preprocess(data)`
        self.save(data) # save output data

flow = d6tflow.Workflow(TaskProcess)
flow.run() # execute task with dependencies
data = flow.outputLoad() # load output data
```

Learn more about *Writing and Managing Tasks* and *Running Workflows*.

7.2.3 Interactive Notebook

Live mybinder example http://tiny.cc/d6tflow-start-interactive

7.2.4 Design Pattern Templates for Machine Learning Workflows

See code templates for a larger real-life project at https://github.com/d6t/d6tflow-template. Clone & code!

7.3 Writing and Managing Tasks

7.3.1 What are tasks?

Tasks are the main object you will be interacting with. They allow you to:

- define input dependency tasks
- **process data**

  - load input data from upstream tasks
  - save output data for downstream tasks
- run tasks
- load output data

You write your own tasks by inheriting from one of the predefined d6tflow task formats, for example pandas dataframes saved to parquet.

```
class YourTask(d6tflow.tasks.TaskPqPandas):
```

Additional details on how to write tasks is below. To run tasks see *Running Workflows*.

7.3.2 Define Upstream Dependency Tasks

You can define input dependencies by using a *@d6tflow.requires* decorator which takes input tasks. You can have no, one or multiple input tasks. This may be required when the decorator shortcut does not work.

```
# no dependency
class TaskSingleInput(d6tflow.tasks.TaskPqPandas):
    #[...]

# single dependency
@d6tflow.requires(TaskSingleOutput)
class TaskSingleInput(d6tflow.tasks.TaskPqPandas):
    #[...]

# multiple dependencies
@d6tflow.requires({'input1':TaskSingleOutput1, 'input2':TaskSingleOutput2})
class TaskMultipleInput(d6tflow.tasks.TaskPqPandas):
    #[...]
```

7.3.3 Process Data

You process data by writing a `run()` function. This function will take input data, process it and save output data.

```
class YourTask(d6tflow.tasks.TaskPqPandas):

    def run(self):
        # load input data
        # process data
        # save data
```

7.3.4 Load Input Data

Input data from upstream dependency tasks can be easily loaded in `run()`

```
# no dependency
class TaskNoInput(d6tflow.tasks.TaskPqPandas):

    def run(self):
        data = pd.read_csv(d6tflow.settings.dirpath/'file.csv') # data/file.csv
```

(continues on next page)

(continued from previous page)

```
# single dependency, single output
@d6tflow.requires(TaskSingleOutput)
class TaskSingleInput(d6tflow.tasks.TaskPqPandas):
    def run(self):
        data = self.inputLoad()

# single dependency, multiple outputs
@d6tflow.requires(TaskMultipleOutput)
class TaskSingleInput(d6tflow.tasks.TaskPqPandas):
    def run(self):
        data1, data2 = self.inputLoad()

# multiple dependencies, single output
@d6tflow.requires({'input1':TaskSingleOutput1, 'input2':TaskSingleOutput2})
class TaskMultipleInput(d6tflow.tasks.TaskPqPandas):
    def run(self):
        data1 = self.inputLoad()['input1']
        data2 = self.inputLoad()['input2']
        # or
        data1 = self.inputLoad(task='input1')
        data2 = self.inputLoad(task='input2')

# multiple dependencies, multiple outputs
@d6tflow.requires({'input1':TaskMultipleOutput1, 'input2':TaskMultipleOutput2})
class TaskMultipleInput(d6tflow.tasks.TaskPqPandas):
    def run(self):
        data = self.inputLoad(as_dict=True)
        data1a = data['input1']['output1']
        data1b = data['input1']['output2']
        data2a = data['input2']['output1']
        data2b = data['input2']['output2']
        # or
        data1a, data1b = self.inputLoad()["input1"]
        data2a, data2b = self.inputLoad()["input2"]
        # or
        data1a, data1b = self.inputLoad(task='input1')
        data2a, data2b = self.inputLoad(task='input2')

# multiple dependencies (without using dictionary), multiple outputs
@d6tflow.requires(TaskMultipleOutput1, TaskMultipleOutput2)
class TaskMultipleInput(d6tflow.tasks.TaskPqPandas):
    def run(self):
        data = self.inputLoad(as_dict=True)
        data1a = data[0]['output1']
        data1b = data[0]['output2']
        data2a = data[1]['output1']
        data2b = data[1]['output2']
        # or
        data1a, data1b = self.inputLoad()[0]
        data2a, data2b = self.inputLoad()[1]
        # or
        data1a, data1b = self.inputLoad(task=0)
        data2a, data2b = self.inputLoad(task=1)
```

Load External Files

You probably want to load external data which is not the output of a task. There are a few options.

```
class TaskExternalData(d6tflow.tasks.TaskPqPandas):

    def run(self):

        import pandas as pd
        # read from d6tflow data folder
        data = pd.read_parquet(d6tflow.settings.dirpath/'file.pq')

        # totally manual
        data = pd.read_parquet('/some/folder/file.pq')

        # multiple files
        from d6tstack.combine_csv import CombinerCSV
        def do_stuff(df):
            return df
        df = CombinerCSV(glob.glob('*.csv'), apply_after_read=do_stuff).to_pandas)
```

For more advanced options see *Sharing Workflows and Outputs*

Dynamic Inputs

See *Dynamic Tasks*

7.3.5 Save Output Data

Saving output data is quick and convenient. You can save a single or multiple outputs.

```
# quick save one output
class TaskSingleOutput(d6tflow.tasks.TaskPqPandas):

    def run(self):
        self.save(data_output)

# save more than one output
class TaskMultipleOutput(d6tflow.tasks.TaskPqPandas):
    persist=['output1','output2'] # declare what you will save

    def run(self):
        self.save({'output1':data1, 'output2':data2}) # needs to match self.persist
```

When you have multiple outputs and don't include `persist` you will get `raise ValueError('Save dictionary needs to consistent with Task.persist')`

Where Is Output Data Saved?

Output data by default is saved in `data/`, you can check with

```
d6tflow.settings.dirpath # folder where workflow output is saved
TaskTrain().output().path # file where task output is saved
```

You can change where data is saved using `d6tflow.set_dir('data/')`. See advanced options for *Sharing Workflows and Outputs* Global Data Path can be also changed by including the `path` parameter to the Workflow.

Changing Task Output Formats

See *Targets*

7.3.6 Running tasks

See *Running Workflows*

7.3.7 Load Output Data

Once a workflow is run and the task is complete, you can easily load its output data by referencing the task.

```
data = flow.outputLoad() # load default task output
data = flow.outputLoad(as_dict=True) # useful for multi output
data2 = flow.outputLoad(TaskMultipleOutput, as_dict=True) # load another task output
data2['data1']
data2['data2']
```

Before you load output data you need to run the workflow. See *run the workflow*. If a task has not been run, it will show

```
raise RuntimeError('Target does not exist, make sure task is complete')
RuntimeError: Target does not exist, make sure task is complete
```

Loading Output Data with Parameters

If you are *using parameters* this is how you load outputs. Make sure you run the task with that parameter first.

```
params = {'default_params':{}, 'use_params':{'preprocess':True}}
flow = d6tflow.WorkflowMulti(TaskSingleOutput, params)
data = flow.outputLoad() # load default task output
data['default_params']
data['use_params']

# multi output
data2 = flow.outputLoad(TaskMultipleOutput, as_dict=True) # load another task output
data2['default_params']['data1']
data2['default_params']['data2']
data2['use_params']['data1']
data2['use_params']['data2']
```

7.3.8 Putting it all together

See full example https://github.com/d6t/d6tflow/blob/master/docs/example-ml.md

See real-life project template https://github.com/d6t/d6tflow-template

7.3.9 Advanced: task attribute overrides

persist: data items to save, see above *external*: do check dependencies, good for sharing tasks without providing code *target_dir*: specify directory *target_ext*: specify extension *save_attrib*: include taskid in filename *pipename*: d6tpipe to save/load to/from

7.4 Running Tasks and Managing Workflows

A workflow object is used to orchestrate tasks and define a task pipeline.

NB: the workflow object is new preferred way of interacting with workflow. Alternatively, legacy workflow describes the old way which might help understand better how everything works.

7.4.1 Define a workflow object

Workflow object can be defined by passing the default task and the parameters for the pipeline. Both the arguments are optional.

```
flow = d6tflow.Workflow(Task1, params)
flow = d6tflow.Workflow(Task1) # use default params
```

Note you want to pass the task definition, not an instantiated task.

```
import tasks
flow = d6tflow.Workflow(tasks.Task1) # yes
flow = d6tflow.Workflow(tasks.Task1()) # no
```

7.4.2 Previewing Task Execution Status

Running a task will automatically run all the upstream dependencies. Before running a workflow, you can preview which tasks will be run.

```
flow.preview() # default task
flow.preview(TaskTrain) # single task
flow.preview([TaskPreprocess,TaskTrain]) # multiple tasks
```

7.4.3 Running Multiple Tasks as Workflows

To run all tasks in a workflow, run the downstream task you want to complete. It will check if all the upstream dependencies are complete and if not it will run them intelligently for you.

```
flow.run() # default task
flow.run(TaskTrain) # single task
flow.run([TaskPreprocess,TaskTrain]) # multiple tasks
```

If your tasks are already complete, they will not rerun. To force rerunning of all tasks but there are better alternatives, see below.

```
flow.run(forced_all_upstream=True, confirm=False) # use flow.reset() instead
```

7.4.4 How is a task marked complete?

Tasks are complete when task output exists. This is typically the existance of a file, database table or cache. See *Task I/O Formats* how task output is stored to understand what needs to exist for a task to be complete.

```
flow.get_task().complete() # status
flow.get_task().output().path # where is output saved?
flow.get_task().output()['output1'].path # multiple outputs
```

Task Completion with Parameters

If a task has parameters, it needs to be run separately for each parameter to be complete when using different parameter settings. The *d6tflow.WorkflowMulti* helps you do that

```
flow = d6tflow.WorkflowMult(Task1, {'flow1':{'preprocess':False},'flow2':{'preprocess
↪':True}})
flow.run() # will run all flow with all parameters
```

Disable Dependency Checks

By default, for a task to be complete, it checks if all dependencies are complete also, not just the task itself. To check if just the task is complete without checking dependencies, set `d6tflow.settings.check_dependencies=False`

```
flow.reset(TaskGetData, confirm=False)
d6tflow.settings.check_dependencies=True # default
flow.preview() # TaskGetData is pending so all tasks are pending
'''
└--[TaskTrain-{'do_preprocess': 'True'} (PENDING)]
   └--[TaskPreprocess-{'do_preprocess': 'True'} (PENDING)]
      └--[TaskGetData-{} (PENDING)]
'''
d6tflow.settings.check_dependencies=False # deactivate dependency checks
flow.preview()
└--[TaskTrain-{'do_preprocess': 'True'} (COMPLETE)]
   └--[TaskPreprocess-{'do_preprocess': 'True'} (COMPLETE)]
      └--[TaskGetData-{} (PENDING)]
d6tflow.settings.check_dependencies=True # set to default
```

7.4.5 Debugging Failures

If a task fails, it will show the stack trace. You need to look further up in the stack trace to find the line that caused the error. You can also set breakpoints in the task obviously.

```
File "tasks.py", line 37, in run => error is here
    1/0
ZeroDivisionError: division by zero

[...] => look further up to find error

===== d6tflow Execution Summary =====
Scheduled 2 tasks of which:
* 1 complete ones were encountered:
```

(continues on next page)

(continued from previous page)

```
    - 1 TaskPreprocess(do_preprocess=True)
* 1 failed:
    - 1 TaskTrain(do_preprocess=True)
This progress looks :( because there were failed tasks
===== d6tflow Execution Summary =====

 File
     raise RuntimeError('Exception found running flow, check trace')
RuntimeError: Exception found running flow, check trace

=> look further up to find error
```

7.4.6 Rerun Tasks When You Make Changes

You have several options to force tasks to reset and rerun. See sections below on how to handle parameter, data and code changes.

```
# preferred way: reset single task, this will automatically run all upstream␣
↪dependencies
flow.reset(TaskGetData, confirm=False) # remove confirm=False to avoid accidentally␣
↪deleting data

# force execution including upstream tasks
flow.run([TaskTrain()],forced_all=True, confirm=False)

# force run everything
flow.run(forced_all_upstream=True, confirm=False)
```

When to reset and rerun tasks?

Typically you want to reset and rerun tasks when:

- parameters changed
- data changed
- code changed

Handling Parameter Change

As long as the parameter is defined in the task, d6tflow will automatically rerun tasks with different parameters.

```
flow = d6tflow.WorkflowMult(Task1, {'flow1':{'preprocess':False},'flow2':{'preprocess
↪':True}})
flow.run() # executes 2 flows, one for each task
```

For d6tflow to intelligently figure out which tasks to rerun, the parameter has to be defined in the task. The downstream task (*TaskTrain*) has to pass on the parameter to the upstream task (*TaskPreprocess*).

```
class TaskGetData(d6tflow.tasks.TaskPqPandas):
# no parameter dependence

class TaskPreprocess(d6tflow.tasks.TaskCachePandas):  # save data in memory
```

(continues on next page)

(continued from previous page)

```
    do_preprocess = d6tflow.BoolParameter(default=True) # parameter for preprocessing␣
↪yes/no

@d6tflow.requires(TaskPreprocess)
class TaskTrain(d6tflow.tasks.TaskPickle):
    # pass parameter upstream
    # no need for to define it again: do_preprocess = d6tflow.
↪BoolParameter(default=True)
```

See [d6tflow docs for handling parameter inheritance](https://d6tflow.readthedocs.io/en/stable/api/d6tflow.util.html#using-inherits-and-requires-to-ease-parameter-pain)

Default Parameter Values in Config

As an alternative to inheriting parameters, you can define defaults in a config files. When you change the config it will automatically rerun tasks.

```
class TaskPreprocess(d6tflow.tasks.TaskCachePandas):
    do_preprocess = d6tflow.BoolParameter(default=cfg.do_preprocess) # store default␣
↪in config
```

Handling Data Change

Premium feature, request access at https://pipe.databolt.tech/gui/request-premium/. You can manually reset tasks if you know your data has changed.

Handling Code Change

Premium feature, request access at https://pipe.databolt.tech/gui/request-premium/. You can manually reset tasks if you know your code has changed.

Forcing a Single Task to Run

You can always run single tasks by calling the *run()* function. This is useful during debugging. However, this will only run this one task and not take care of any downstream dependencies.

```
# forcing execution
flow.get_task().run()
# or
TaskTrain().run()
```

7.4.7 Hiding Execution Output

By default, the workflow execution summary is shown, because it shows important information which tasks were run and if any failed. At times, eg during deployment, it can be desirable to not show the execution output.

```
d6tflow.settings.execution_summary = False # global
# or
flow.run(execution_summary=False) # at each run
```

While typically not necessary, you can control change the log level to see additional log data. Default is `WARNING`. It is a global setting, modify before you execute `d6tflow.run()`.

```
d6tflow.settings.log_level = 'WARNING' # 'DEBUG', 'INFO', 'WARNING', 'ERROR',
↪'CRITICAL'
```

7.5 Task I/O Targets

7.5.1 How is task data saved and loaded?

Task data is saved in a file, database table or memory (cache). You can control how task output data is saved by chosing the right parent class for a task. In the example below, data is saved as parquet and loaded as a pandas dataframe because the parent class is `TaskPqPandas`. The python object you want to save determines how you can save the data.

```
class YourTask(d6tflow.tasks.TaskPqPandas):
```

Task Output Location

By default file-based task output is saved in `data/`. You can customize where task output is saved.

```
d6tflow.set_dir('../data')
```

7.5.2 Core task targets (Pandas)

What kind of object you want to save determines which Task class you need to use.

- **pandas**
  - `d6tflow.tasks.TaskPqPandas`: save to parquet, load as pandas
  - `d6tflow.tasks.TaskCachePandas`: save to memory, load as pandas
  - `d6tflow.tasks.TaskCSVPandas`: save to CSV, load as pandas
  - `d6tflow.tasks.TaskExcelPandas`: save to Excel, load as pandas
  - `d6tflow.tasks.TaskSQLPandas`: save to SQL, load as pandas (premium, see below)
- **dicts**
  - `d6tflow.tasks.TaskJson`: save to JSON, load as python dict
  - `d6tflow.tasks.TaskPickle`: save to pickle, load as python list
  - **NB**: don't save a dict of pandas dataframes as pickle, instead save as multiple outputs, see "save more than one output" in *Tasks*
- **any python object (eg trained models)**
  - `d6tflow.tasks.TaskPickle`: save to pickle, load as python list
  - `d6tflow.tasks.TaskCache`: save to memory, load as python object
- dask, SQL, pyspark: premium features, see below

7.5.3 Premium Targets (Dask, SQL, Pyspark)

Database Targets

d6tflow premium has database targets, request access at https://pipe.databolt.tech/gui/request-premium/

Dask Targets

d6tflow premium has dask targets, request access at https://pipe.databolt.tech/gui/request-premium/

Pyspark Targets

d6tflow premium has pyspark targets, request access at https://pipe.databolt.tech/gui/request-premium/

7.5.4 Community Targets

Keras Model Targets

For saving Keras model targets

```
from d6tflow.tasks.h5 import TaskH5Keras
```

7.5.5 Writing Your Own Targets

This is often relatively simple since you mostly need to implement *load()* and *save()* functions. For more advanced cases you also have to implement *exist()* and *invalidate()* functions. Check the source code for details or raise an issue.

7.6 Sharing Workflows and Outputs

7.6.1 Introduction

With d6tflow you can Export and Import Tasks from other projects and files. This makes sharing flow output and handing projects off to others very seamless. Some cases when you want to do this include:

- data engineers share data with data scientists
- vendors sharing data with clients
- teachers sharing data with students

7.6.2 Exporting Tasks

You can Export your tasks into a new File or print the tasks in the console. All parameters, paths, task_group will be exported.

```
class Task1():
    def run(self):
        #Save

@d6tflow.requires(Task1)
class Task2():
    def run(self):
        #Save

flow = d6tflow.Workflow(Task2())

# This will only export Task 2 to console
e = d6tflow.FlowExport(tasks=Task2())
e.generate()

# This will export All the flow (Task1, Task2) to a file
e = d6tflow.FlowExport(flows=flow, save=True, path_export='tasks_export.py')
e.generate()
```

7.6.3 Attaching Flows

In more complex projects, users need to import data from many sources. Flows can be attached together in order to access the data generated in one flow inside the other.

```
class Task1():
    def run(self):
        #Save

@d6tflow.requires(Task1)
class Task2():
    def run(self):
        #Save

class Task3():
    def run():
        temp_flow_df = self.flows['flow'].outputLoad()
        self.save(temp_flow_df)

# Define Both flows and run the first
flow = d6tflow.workflow(Task1)
flow2 = d6tflow.workflow(Task3)
flow.run()

# Attach the First Flow to the Second
flow2.attach(flow, 'flow')
flow2.run()
```

7.7 Advanced: Dynamic Tasks

Sometimes you might not know exactly what other tasks to depend on until runtime. There are several cases of dynamic dependencies.

7.7.1 Fixed Dynamic

If you have a fixed set parameters, you can make *requires()* "dynamic".

```
class TaskInput(d6tflow.tasks.TaskPqPandas):
    param = d6tflow.Parameter()
    ...

class TaskYieldFixed(d6tflow.tasks.TaskPqPandas):

    def requires(self):
        return dict([(s,TaskInput(param=s)) for s in ['a','b','c']])

    def run(self):
        df = self.inputLoad()
        df = pd.concat(df)
```

You could also use this to load an unknown number of files as a starting point for the workflow.

```
def requires(self):
    return dict([(s,TaskInput(param=s)) for s in glob.glob('*.csv')])
```

7.7.2 Collector Task

If you want to spawn multiple tasks without processing any of the outputs, you can use *TaskAggregator*. This task should do nothing but yield other tasks.

```
@d6tflow.requires(TrainModel1,TrainModel2) # inherit all params from input tasks
class TrainAllModels(d6tflow.tasks.TaskAggregator):

    def run(self):
        yield self.clone(TrainModel1)
        yield self.clone(TrainModel2)
```

Alternatively, you can achieve the same using the *WorkflowMulti* object with additional flexibility.

```
params = dict()
params_all = d6tflow.utils.params_generator_single({'param':['a','b']},params)

flow = d6tflow.WorkflowMulti(tasks_search.SearchModelTrain, params=params_all)
flow.run()
```

If you want to run the workflow with multiple parameters at the same time, you can use *TaskAggregator* to yield multiple tasks.

```
class TaskAggregator(d6tflow.tasks.TaskAggregator):

    def run(self):
        yield TaskTrain(do_preprocess=False)
        yield TaskTrain(do_preprocess=True)
```

7.7.3 Fully Dynamic

This doesn't work yet... It's actually quite rare though that you need that though. Parameters normally fall in a fixed range which can be solved with the approaches above. Another typical reason you would want to do this is to load an

unknown number of input files which you can do manually, see "Load External Files" in *tasks*.

```
class TaskA(d6tflow.tasks.TaskCache):
    param = d6tflow.IntParameter()
    def run(self):
        self.save(self.param)

class TaskB(d6tflow.tasks.TaskCache):
    param = d6tflow.IntParameter()

    def requires(self):
        return TaskA()

    def run(self):
        value = 1
        df_train = self.input(param=value).load()
```

7.8 Advanced: Parameters

Intelligent parameter management is one of the most powerful features of d6tflow. New users often have questions on parameter management, this is an important section to read.

7.8.1 Specifying parameters

Tasks can take any number of parameters.

```
import datetime

class TaskTrain(d6tflow.tasks.TaskPqPandas):
    do_preprocess = d6tflow.BoolParameter(default=True)
    model = d6tflow.Parameter(default='xgboost')
```

7.8.2 Running tasks with parameters

Just pass the parameters values, everything else is the same.

```
d6tflow.run(TaskTrain() # use default do_preprocess=True, model='xgboost'
d6tflow.run(TaskTrain(do_preprocess=False, model='nnet')) # specify non-default␣
↪parameters
# or
params = dict(do_preprocess=False, model='nnet')
d6tflow.run(TaskTrain(**params)) # specify non-default parameters
```

Note that you can pass parameters for upstream tasks directly to the terminal task, they will be automatically passed to upstream tasks. See below for details.

7.8.3 Loading Output Data with Parameters

If you are *using parameters* this is how you load outputs. Make sure you run the task with that parameter first.

```
df = TaskTrain().output().load() # load data with default parameters
params = dict(do_preprocess=False, model='nnet')
df = TaskTrain(**params).output().load() # specify non-default parameters
```

7.8.4 Parameter types

Parameters can be typed.

```
import datetime

class TaskTrain(d6tflow.tasks.TaskPqPandas):
    do_preprocess = d6tflow.BoolParameter(default=True)
    dt_start = d6tflow.DateParameter(default=datetime.date(2010,1,1))
    dt_end = d6tflow.DateParameter(default=datetime.date(2020,1,1))

    def run(self):
        if self.do_preprocess:
            if self.dt_start>datetime.date(2010,1,1):
                pass
```

Overview https://d6tflow.readthedocs.io/en/stable/parameters.html#parameter-types

Full reference https://d6tflow.readthedocs.io/en/stable/api/d6tflow.parameter.html

7.8.5 Avoid repeating parameters in every class

You often need to pass parameters between classes. With d6tflow, you do not need to repeat parameters in every class, they are automatically managed, that is they are automatically passed to upstream tasks from downstream tasks.

```
class TaskTrain(d6tflow.tasks.TaskPqPandas):
    do_preprocess = d6tflow.BoolParameter(default=True)
    dt_start = d6tflow.DateParameter(default=datetime.date(2010,1,1))
    dt_end = d6tflow.DateParameter(default=datetime.date(2020,1,1))
    # ...

@d6tflow.requires(TaskTrain) # automatically inherits parameters
class TaskEvaluate(d6tflow.tasks.TaskPickle):

    # requires() is automatic
    # do_preprocess => inherited from TaskTrain
    # dt_start => inherited from TaskTrain
    # dt_end => inherited from TaskTrain

    def run(self):
        print(self.do_preprocess) # inherited
        print(self.dt_start) # inherited

d6tflow.preview(TaskEvaluate(do_preprocess=False))  # specify non-default parameters
'''
└--[TaskEvaluate-{'do_preprocess': 'False', 'dt_start': '2010-01-01', 'dt_end':
↪'2020-01-01'} (PENDING)]
└--[TaskTrain-{'do_preprocess': 'False', 'dt_start': '2010-01-01', 'dt_end': '2020-
↪01-01'} (PENDING)] => automatically passed upstream
'''
```

Note that you can pass parameters for upstream tasks directly to the terminal task, they will be automatically passed to upstream tasks. *do_preprocess=False* will be passed down from *TaskEvaluate* to *TaskTrain*.

If you require multiple tasks, you can inherit parameters from those tasks. *TaskEvaluate* depeonds on both *TaskTrain* and *TaskPredict*.

```
class TaskTrain(d6tflow.tasks.TaskPqPandas):
    do_preprocess = d6tflow.BoolParameter(default=True)

class TaskPredict(d6tflow.tasks.TaskPqPandas):
    dt_start = d6tflow.DateParameter(default=datetime.date(2010,1,1))
    dt_end = d6tflow.DateParameter(default=datetime.date(2020,1,1))

@d6tflow.requires(TaskTrain,TaskPredict) # inherit all params from input tasks
class TaskEvaluate(d6tflow.tasks.TaskPickle):
    # do_preprocess => inherited from TaskTrain
    # dt_start => inherited from TaskPredict
    # dt_end => inherited from TaskPredict

    def run(self):
        print(self.do_preprocess) # inherited from TaskTrain
        print(self.dt_start) # inherited from TaskPredict

d6tflow.preview(TaskEvaluate(do_preprocess=False))  # specify non-default parameters
'''
└--[TaskEvaluate-{'do_preprocess': 'False', 'dt_start': '2010-01-01', 'dt_end':
↪'2020-01-01'} (PENDING)]
   |--[TaskTrain-{'do_preprocess': 'False'} (PENDING)] => automatically passed␣
↪upstream
   └--[TaskPredict-{'dt_start': '2010-01-01', 'dt_end': '2020-01-01'} (PENDING)] =>␣
↪automatically passed upstream
'''
```

@d6tflow.requires also works with aggregator tasks.

```
@d6tflow.requires(TaskTrain,TaskPredict) # inherit all params from input tasks
class TaskEvaluate(d6tflow.tasks.TaskAggregator):

    def run(self):
        yield self.clone(TaskTrain)
        yield self.clone(TaskPredict)
```

For another ML example see https://github.com/d6t/d6tflow/blob/master/docs/example-ml.md

For more details see https://d6tflow.readthedocs.io/en/stable/api/d6tflow.util.html

The project template also implements task parameter inheritance https://github.com/d6t/d6tflow-template

7.8.6 Avoid repeating parameters when referring to tasks

To run tasks and load their output for different parameters, you have to pass them to the task. Instead of hardcoding them each time, it is best to keep them in a dictionary and pass that to the task.

```
# avoid this
d6tflow.run(TaskTrain(do_preprocess=False, model='nnet'))
TaskTrain(do_preprocess=False, model='nnet').outputLoad()
```

(continues on next page)

(continued from previous page)

```
# better
params = dict(do_preprocess=False, model='nnet')
d6tflow.run(TaskTrain(**params))
TaskTrain(**params).outputLoad()
```

7.9 d6tflow

7.9.1 d6tflow package

Submodules

Module contents

class d6tflow.**FlowExport**(*tasks=None*, *flows=None*, *save=False*, *path_export='tasks_d6tpipe.py'*)

Bases: object

Auto generate task files to quickly share workflows with others using d6tpipe.

Parameters

- **tasks** (*obj*) – task or list of tasks to share
- **flows** (*obj*) – flow or list of flows to get tasks from.
- **save** (*bool*) – save to tasks file
- **path_export** (*str*) – filename for tasks to export.

generate()

Generate output files

class d6tflow.**Workflow**(*task=None*, *params=None*, *path=None*, *env=None*)

Bases: object

The class is used to orchestrate tasks and define a task pipeline

attach_flow(*flow=None*, *flow_name='flow'*)

complete(*task=None*)

get_task(*task=None*)

Get task with the workflow parameters

Parameters **task** (*class*) –

Retuns: An instance of task class with the workflow parameters

output(*task=None*)

outputLoad(*task=None*, *keys=None*, *as_dict=False*, *cached=False*)

Load output from task with the workflow parameters

Parameters

- **task** (*class*) – task class
- **keys** (*list*) – list of data to load
- **as_dict** (*bool*) – cache data in memory
- **cached** (*bool*) – cache data in memory

Returns: list or dict of all task output

outputLoadAll(*task=None*, *keys=None*, *as_dict=False*, *cached=False*)
Load all output from task with the workflow parameters

Parameters

- **task** (*class*) – task class
- **keys** (*list*) – list of data to load
- **as_dict** (*bool*) – cache data in memory
- **cached** (*bool*) – cache data in memory

Returns: list or dict of all task output

outputLoadMeta(*task=None*)

outputPath(*task=None*)
Ouputs the Path given a task

Parameters **task** (*class*) – task class

Returns: list or dict of all task paths

preview(*tasks=None*, *indent=''*, *last=True*, *show_params=True*, *clip_params=False*)
Preview task flows with the workflow parameters

Parameters **tasks** (*class, list*) – task class or list of tasks class

reset(*task=None*, *confirm=False*)

reset_downstream(*task*, *task_downstream=None*, *confirm=False*)
Invalidate all downstream tasks in a flow.

For example, you have 3 dependant tasks. Normally you run Task3 but you've changed parameters for Task1. By invalidating Task3 it will check the full DAG and realize Task1 needs to be invalidated and therefore Task2 and Task3 also.

Parameters

- **task** (*obj*) – task to invalidate. This should be an downstream task for which you want to check downstream dependencies for invalidation conditions
- **task_downstream** (*obj*) – downstream task target
- **confirm** (*bool*) – confirm operation

reset_upstream(*task*, *confirm=False*)

run(*tasks=None*, *forced=None*, *forced_all=False*, *forced_all_upstream=False*, *confirm=False*, *workers=1*, *abort=True*, *execution_summary=None*, ***kwargs*)
Run tasks with the workflow parameters. See luigi.build for additional details

Parameters

- **tasks** (*class, list*) – task class or list of tasks class
- **forced** (*list*) – list of forced tasks
- **forced_all** (*bool*) – force all tasks
- **forced_all_upstream** (*bool*) – force all tasks including upstream
- **confirm** (*list*) – confirm invalidating tasks
- **workers** (*int*) – number of workers

- **abort** (*bool*) – on errors raise exception
- **execution_summary** (*bool*) – print execution summary
- **kwargs** – keywords to pass to luigi.build

set_default(*task*)
Set default task for the workflow object

Parameters **task** (*obj*) –

class d6tflow.**WorkflowMulti**(*task=None*, *params=None*, *path=None*, *env=None*)
Bases: object

A multi experiment workflow can be defined with multiple flows and separate parameters for each flow and a default task. It is mandatory to define the flows and parameters for each of the flows.

get_flow(*flow*)
Get flow by name

Parameters **flow** (*string*) – The name of the experiment for which the flow is to be run. If nothing is passed, all the flows are run

Retuns: An instance of Workflow

get_task(*task=None*, *flow=None*)
Get task with the workflow parameters for a flow

Parameters

- **flow** (*string*) – The name of the experiment for which the flow is to be run. If nothing is passed, all the flows are run
- **task** (*class*) – task class

Retuns: An instance of task class with the workflow parameters

outputLoad(*task=None*, *flow=None*, *keys=None*, *as_dict=False*, *cached=False*)
Load output from task with the workflow parameters for a flow

Parameters

- **flow** (*string*) – The name of the experiment for which the flow is to be run. If nothing is passed, all the flows are run
- **task** (*class*) – task class
- **keys** (*list*) – list of data to load
- **as_dict** (*bool*) – cache data in memory
- **cached** (*bool*) – cache data in memory

Returns: list or dict of all task output

outputLoadAll(*task=None*, *flow=None*, *keys=None*, *as_dict=False*, *cached=False*)
Load all output from task with the workflow parameters for a flow

Parameters

- **flow** (*string*) – The name of the experiment for which the flow is to be run. If nothing is passed, all the flows are run
- **task** (*class*) – task class
- **keys** (*list*) – list of data to load
- **as_dict** (*bool*) – cache data in memory

- **cached** (*bool*) – cache data in memory

Returns: list or dict of all task output

outputLoadMeta(*task=None*, *flow=None*)

outputPath(*task=None*, *flow=None*)
Ouputs the Path given a task

Parameters

- **task** (*class*) – task class
- **flow** (*string*) – The name of the experiment for which the flow is to be run. If nothing is passed, all the flows are run

Returns: list or dict of all task paths

preview(*tasks=None*, *flow=None*, *indent=''*, *last=True*, *show_params=True*, *clip_params=False*)
Preview task flows with the workflow parameters for a flow

Parameters

- **flow** (*string*) – The name of the experiment for which the flow is to be run. If nothing is passed, all the flows are run
- **tasks** (*class, list*) – task class or list of tasks class

reset(*task=None*, *flow=None*, *confirm=False*)

reset_downstream(*task=None*, *flow=None*, *confirm=False*)

reset_upstream(*task=None*, *flow=None*, *confirm=False*)

run(*tasks=None*, *flow=None*, *forced=None*, *forced_all=False*, *forced_all_upstream=False*, *confirm=False*, *workers=1*, *abort=True*, *execution_summary=None*, ***kwargs*)
Run tasks with the workflow parameters for a flow. See luigi.build for additional details

Parameters

- **flow** (*string*) – The name of the experiment for which the flow is to be run. If nothing is passed, all the flows are run
- **tasks** (*class, list*) – task class or list of tasks class
- **forced** (*list*) – list of forced tasks
- **forced_all** (*bool*) – force all tasks
- **forced_all_upstream** (*bool*) – force all tasks including upstream
- **confirm** (*list*) – confirm invalidating tasks
- **workers** (*int*) – number of workers
- **abort** (*bool*) – on errors raise exception
- **execution_summary** (*bool*) – print execution summary
- **kwargs** – keywords to pass to luigi.build

set_default(*task*)
Set default task for the workflow. The default task is set for all the experiments

Parameters **task** (*obj*) –

d6tflow.**clone_parent**(*cls*)

class d6tflow.**dict_inherits**(**tasks_to_inherit*)
Bases: object

class d6tflow.**dict_requires**(**tasks_to_require*)
Bases: object

d6tflow.**inherits**(**tasks_to_inherit*)

d6tflow.**invalidate_all**(*confirm=False*)
Invalidate all tasks by deleting all files in data directory

Parameters **confirm** (*bool*) – confirm operation

d6tflow.**invalidate_downstream**(*task*, *task_downstream*, *confirm=False*)
Invalidate all downstream tasks in a flow.

For example, you have 3 dependant tasks. Normally you run Task3 but you've changed parameters for Task1. By invalidating Task3 it will check the full DAG and realize Task1 needs to be invalidated and therefore Task2 and Task3 also.

Parameters

- **task** (*obj*) – task to invalidate. This should be an downstream task for which you want to check downstream dependencies for invalidation conditions
- **task_downstream** (*obj*) – downstream task target
- **confirm** (*bool*) – confirm operation

d6tflow.**invalidate_orphans**(*confirm=False*)
Invalidate all unused task outputs

Parameters **confirm** (*bool*) – confirm operation

d6tflow.**invalidate_upstream**(*task*, *confirm=False*)
Invalidate all tasks upstream tasks in a flow.

For example, you have 3 dependant tasks. Normally you run Task3 but you've changed parameters for Task1. By invalidating Task3 it will check the full DAG and realize Task1 needs to be invalidated and therefore Task2 and Task3 also.

Parameters

- **task** (*obj*) – task to invalidate. This should be an upstream task for which you want to check upstream dependencies for invalidation conditions
- **confirm** (*bool*) – confirm operation

d6tflow.**requires**(**tasks_to_require*)

d6tflow.**set_dir**(*dir=None*)
Initialize d6tflow

Parameters **dir** (*str*) – data output directory

d6tflow.**show**(*task*)
Show task execution status

Parameters **tasks** (*obj, list*) – task or list of tasks

d6tflow.**taskflow_downstream**(*task*, *task_downstream*, *only_complete=False*)
Get all downstream outputs for a task

Parameters

- **task** (*obj*) – task

- **task_downstream** (*obj*) – downstream target task

d6tflow.**taskflow_upstream**(*task*, *only_complete=False*)
Get all upstream inputs for a task

Parameters **task** (*obj*) – task

d6tflow.tasks module

class d6tflow.tasks.**TaskAggregator**(**args*, ***kwargs*)
Bases: luigi.task.Task

Task which yields other tasks

NB: Use this function by implementing *run()* which should do nothing but yield other tasks

example:

```
class TaskCollector(d6tflow.tasks.TaskAggregator):
    def run(self):
        yield Task1()
        yield Task2()
```

complete(*cascade=True*)
If the task has any outputs, return `True` if all outputs exist. Otherwise, return `False`.

However, you may freely override this method with custom logic.

invalidate(*confirm=True*)

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

outputLoad(*keys=None*, *as_dict=False*, *cached=False*)

reset(*confirm=True*)

class d6tflow.tasks.**TaskCSVGZPandas**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to CSV

target_class
alias of *d6tflow.targets.CSVGZPandasTarget*

target_ext = 'csv.gz'

class d6tflow.tasks.**TaskCSVPandas**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to CSV

target_class
alias of *d6tflow.targets.CSVPandasTarget*

target_ext = 'csv'

class d6tflow.tasks.**TaskCache**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to cache

target_class
alias of *d6tflow.targets.CacheTarget*

target_ext = 'cache'

class d6tflow.tasks.**TaskCachePandas**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to cache pandas dataframes

target_class
alias of *d6tflow.targets.PdCacheTarget*

target_ext = 'cache'

class d6tflow.tasks.**TaskData**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: luigi.task.Task

Task which has data as input and output

Parameters

- **target_class** (*obj*) – target data format
- **target_ext** (*str*) – file extension
- **persist** (*list*) – list of string to identify data
- **data** (*dict*) – data container for all outputs

complete(**args*, ***kwargs*)

classmethod get_param_values(*params*, *args*, *kwargs*)
Get the values of the parameters from the args and kwargs.

Parameters

- **params** – list of (param_name, Parameter).
- **args** – positional arguments
- **kwargs** – keyword arguments.

Returns list of *(name, value)* tuples, one for each parameter.

get_pipe()
Get associated pipe object

get_pipename(**args*, ***kwargs*)

inputLoad(*keys=None*, *task=None*, *cached=False*, *as_dict=False*)
Load all or several outputs from task

Parameters

- **keys** (*list*) – list of data to load
- **task** (*str*) – if requires multiple tasks load that task 'input1' for eg *def requires: {'input1':Task1(), 'input2':Task2()}*
- **cached** (*bool*) – cache data in memory

- **as_dict** (*bool*) – if the inputs were saved as a dictionary. use this to return them as dictionary.

Returns: list or dict of all task output

invalidate(*confirm=True*)
Reset a task, eg by deleting output file

metaLoad()

metaSave(*data*)

metadata = None

output()
Similar to luigi task output

outputLoad(*keys=None*, *as_dict=False*, *cached=False*)
Load all or several outputs from task

Parameters

- **keys** (*list*) – list of data to load
- **as_dict** (*bool*) – cache data in memory
- **cached** (*bool*) – cache data in memory

Returns: list or dict of all task output

outputLoadAllMeta()

outputLoadMeta()

persist = ['data']

pull(***kwargs*)
Pull files from data repo

pull_preview(***kwargs*)
Preview pull files from data repo

push(***kwargs*)
Push files to data repo

push_preview(***kwargs*)
Preview push files to data repo

reset(*confirm=True*)
Reset a task, eg by deleting output file

save(*data*, ***kwargs*)
Persist data to target

Parameters **data** (*dict*) – data to save. keys are the self.persist keys and values is data

saveMeta(*data*)

target_class
alias of *d6tflow.targets.DataTarget*

target_ext = 'ext'

class d6tflow.tasks.**TaskExcelPandas**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to Excel

target_class
alias of *d6tflow.targets.ExcelPandasTarget*

target_ext = 'xlsx'

class d6tflow.tasks.**TaskJson**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to json

target_class
alias of *d6tflow.targets.JsonTarget*

target_ext = 'json'

class d6tflow.tasks.**TaskPickle**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to pickle

target_class
alias of *d6tflow.targets.PickleTarget*

target_ext = 'pkl'

class d6tflow.tasks.**TaskPqPandas**(**args*, *path=None*, *flows=None*, ***kwargs*)
Bases: *d6tflow.tasks.TaskData*

Task which saves to parquet

target_class
alias of *d6tflow.targets.PqPandasTarget*

target_ext = 'parquet'

d6tflow.targets module

class d6tflow.targets.**CSVGZPandasTarget**(*path=None*)
Bases: *d6tflow.targets.CSVPandasTarget*

Saves to CSV gzip, loads to pandas dataframe

save(**args*, ***kwargs*)

class d6tflow.targets.**CSVPandasTarget**(*path=None*)
Bases: *d6tflow.targets.DataTarget*

Saves to CSV, loads to pandas dataframe

load(*cached=False*, ***kwargs*)
Load from csv to pandas dataframe

Parameters

- **cached** (*bool*) – keep data cached in memory
- ****kwargs** – arguments to pass to pd.read_csv

Returns: pandas dataframe

save(**args*, ***kwargs*)

class d6tflow.targets.**CacheTarget**(*path=None*, *format=None*, *is_tmp=False*)
Bases: luigi.local_target.LocalTarget

Saves to in-memory cache, loads to python object

exists()
Returns `True` if the path for this FileSystemTarget exists; `False` otherwise.

This method is implemented by using `fs`.

invalidate()

load(*cached=True*)
Load from in-memory cache

Returns: python object

save(**args*, ***kwargs*)

class d6tflow.targets.**DataTarget**(*path=None*)
Bases: d6tflow.targets._LocalPathTarget

Local target which saves in-memory data (eg dataframes) to persistent storage (eg files) and loads from storage to memory

This is an abstract class that you should extend.

load(*fun*, *cached=False*, ***kwargs*)
Runs a function to load data from storage into memory

Parameters

- **fun** (*function*) – loading function
- **cached** (*bool*) – keep data cached in memory
- ****kwargs** – arguments to pass to *fun*

Returns: data object

save(*df*, *fun*, ***kwargs*)
Runs a function to save data from memory into storage

Parameters

- **df** (*obj*) – data to save
- **fun** (*function*) – saving function
- ****kwargs** – arguments to pass to *fun*

Returns: filename

class d6tflow.targets.**ExcelPandasTarget**(*path=None*)
Bases: *d6tflow.targets.DataTarget*

Saves to Excel, loads to pandas dataframe

load(*cached=False*, ***kwargs*)
Load from Excel to pandas dataframe

Parameters

- **cached** (*bool*) – keep data cached in memory
- ****kwargs** – arguments to pass to pd.read_csv

Returns: pandas dataframe

save(**args*, ***kwargs*)

class d6tflow.targets.**JsonTarget**(*path=None*)
Bases: *d6tflow.targets.DataTarget*

Saves to json, loads to dict

load(*cached=False*, ***kwargs*)
Load from json to dict

Parameters

- **cached** (*bool*) – keep data cached in memory
- ****kwargs** – arguments to pass to json.load

Returns: dict

save(**args*, ***kwargs*)

class d6tflow.targets.**PdCacheTarget**(*path=None*, *format=None*, *is_tmp=False*)
Bases: *d6tflow.targets.CacheTarget*

class d6tflow.targets.**PickleTarget**(*path=None*)
Bases: *d6tflow.targets.DataTarget*

Saves to pickle, loads to python obj

load(*cached=False*, ***kwargs*)
Load from pickle to obj

Parameters

- **cached** (*bool*) – keep data cached in memory
- ****kwargs** – arguments to pass to pickle.load

Returns: dict

save(**args*, ***kwargs*)

class d6tflow.targets.**PqPandasTarget**(*path=None*)
Bases: *d6tflow.targets.DataTarget*

Saves to parquet, loads to pandas dataframe

load(*cached=False*, ***kwargs*)
Load from parquet to pandas dataframe

Parameters

- **cached** (*bool*) – keep data cached in memory
- ****kwargs** – arguments to pass to pd.read_parquet

Returns: pandas dataframe

save(**args*, ***kwargs*)

d6tflow.pipes module

d6tflow.pipes.**all_pull**(*task*, ***kwargs*)
Pull for all upstream tasks in a flow

d6tflow.pipes.**all_pull_preview**(*task*, ***kwargs*)
Pull preview for all upstream tasks in a flow

d6tflow.pipes.**all_push**(*task*, ***kwargs*)
Push for all upstream tasks in a flow

d6tflow.pipes.**all_push_preview**(*task*, ***kwargs*)
Push preview for all upstream tasks in a flow

d6tflow.pipes.**get_dirpath**(*name=None*)
Get a pipe directory as Pathlib.Path

Parameters **name** (*str*) – name of pipe

d6tflow.pipes.**get_pipe**(*name=None*)
Get a pipe

Parameters **name** (*str*) – name of pipe

Returns pipe object

Return type obj

d6tflow.pipes.**init**(*default_pipe_name*, *profile=None*, *local_pipe=False*, *local_api=False*, *reset=False*, *api=None*, *set_dir=True*, *api_args=None*, *pipe_args=None*)
Initialize d6tpipe

Parameters

- **default_pipe_name** (*str*) – name of pipe to store results. Override by setting Task.pipe attribute
- **profile** (*str*) – name of d6tpipe profile to get api if api not provided
- **local_pipe** (*bool*) – use *PipeLocal()*
- **local_api** (*bool*) – use *APILocal()*
- **reset** (*bool*) – reset api and pipe connection
- **api** (*obj*) – d6tpipe api object. if not provided will be loaded
- **set_dir** (*bool*) – if True, set d6tflow directory to default pipe directory
- **api_args** (*dir*) – arguments to pass to api
- **pipe_args** (*dir*) – arguments to pass to pipe

d6tflow.pipes.**init_pipe**(*name=None*, ***kwargs*)

d6tflow.pipes.**set_default**(*name*)
Set default pipe. Will also change d6tflow directory

Parameters **name** (*str*) – name of pipe

d6tflow.functional module

class d6tflow.functional.**Workflow**
Bases: object

Functional Flow class that acts as a manager of all flow steps. Defines all the decorators that can be used on flow functions.

add_global_params(***params*)
Adds params to flow functions. More like declares the params for further use. :param params: dictionary of param name and param type :type params: dict

Example

flow.add_params({'multiplier': d6tflow.IntParameter(default=0)})

delete (*func_to_reset*, **args*, ***kwargs*)
Possibly dangerous! *delete(func)* will delete *all files* in the *data/func* directory of the given func. Useful if you want to delete all function related outputs. Consider using *reset(func, params)* to reset a specific func

deleteAll (**args*, ***kwargs*)
Possibly dangerous! Will delete all files in the *data/* directory of the functions attached to the workflow object. Useful if you want to delete all outputs even the once previously run. Consider using *resetAll()* if you want to only reset the functions with params you have run thus far

outputLoad (*func_to_run*, **args*, ***kwargs*)
Loads all or several outputs from flow step.

Parameters

- **func_to_run** – flow step function
- **keys** (*list*) – list of data to load
- **as_dict** (*bool*) – cache data in memory
- **cached** (*bool*) – cache data in memory

Returns: list or dict of all task output

outputLoadAll (*func_to_run*, **args*, ***kwargs*)
Loads all output from flow task and its parents.

Parameters

- **func_to_run** – flow step function
- **keys** (*list*) – list of data to load
- **as_dict** (*bool*) – cache data in memory
- **cached** (*bool*) – cache data in memory

Returns: list or dict of all task output

params (**args*, ***kwargs*)

persists (**args*, ***kwargs*)

preview (*func_to_preview*, *params: dict*)

requires (**args*, ***kwargs*)

reset (*func_to_reset*, *params=None*, **args*, ***kwargs*)
Resets a particular function. Use with *params* to reset function with the given parameters. If *params* is not used, *reset(func)* will reset the function with all the parameters run thus far

resetAll (**args*, ***kwargs*)
Resets all functions that are attached to the workflow object that have run at least once.

run (**args*, ***kwargs*)

task (**args*, ***kwargs*)

7.10 Functional Tasks

7.10.1 What are functional tasks?

Functional tasks are meant to provide a nice decorator based way of defining tasks.

7.10.2 How to create a functional task?

For defining our tasks we will need to first define a *Workflow()* object.

```
from d6tflow.functional import Workflow
flow = Workflow()
```

Each function is decorated with a *flow.task* decorator - that takes a *d6tflow.tasks.TaskName* as parameter

```
@flow.task(d6tflow.tasks.TaskPqPandas)
def your_functional_task(task):
    print("Running a complicated task!!")
```

You might have noticed we provide a *task* parameter to the function above.

This is deliberate.

If you have worked with d6tflow.task before you would remember having a *self* parameter passed to *run()* method.

Here *task* is exactly that. It contains all methods available in *d6tflow.task.Task*

7.10.3 Running a functional task

All functional tasks are run as *d6tflow.task* under the hood.

So we require to run them as you would run any *d6tflow.task*

Workflow() object comes with a run method which does exactly that.

```
flow.run(your_functional_task)
```

Below is a minimal example of functional task that encompasses everything mentioned above.

```
import d6tflow
from d6tflow.functional import Workflow
import pandas as pd

flow = Workflow()

@flow.task(d6tflow.tasks.TaskCache)
def sample_functional_task(task):
    df = pd.DataFrame({'a':range(3)})
    print("Functional task running!")
    task.save(df)

flow.run(sample_functional_task)
```

7.10.4 Additional decorators

These decorators are to be decorated after @flow.task

- ***@flow.persists***
  - Takes in a list of variables that need to be persisted for the flow task.
  - ```
    @flow.persists(['a1', 'a2'])
    ```
- ***@flow.params***
  - Takes in keyword-arguments of parameters and their types to be used in the function body.
  - ```
    @flow.params(example_argument=d6tflow.IntParameter(default=42))
    ```
- ***@flow.requires***
  - Defines dependencies between flow tasks.
  - ```
    @flow.requires({"foo": func1, "bar": func2})
    @flow.requires(func1)
    ```

Example -

```
...
@flow.task(d6tflow.tasks.TaskCache)
@flow.requires({"a":get_data1, "b":get_data2})
@flow.persists(['aa'])
def example_function(task):
    df = task.inputLoad()
    a = df["a"]
    b = df["b"]
    print(a,b)
    output = pd.DataFrame({'a':range(4)})
    task.save({'aa':output})
...
```

7.10.5 Passing parameters to the *run()* method

We saw in one of the above section how to run functional tasks.

d6tflow also allows you to pass in parameters to these functions dynamically using *@flow.params()*

Below is an example of passing a 'multiplier' paramter to a functional task.

```
@flow.params(multiplier=d6tflow.IntParameter(default=0))
def print_parameter(task):
    print(task.multiplier)

flow.run(print_parameter, params={'multiplier':42})
```

So basically, you define the parameter name and its type with *@flow.params* and then use the *run()* method's *params* to pass in the actual value

7.10.6 Additional methods

Some of the functions that are in d6tflow are available in the *Workflow()* object too!
```

Here's a list of them -

- preview(function)
- outputLoad(function)
- run(functions_as_list)
- reset(function)
- outputLoadAll()

Wait! There is more! Here are some more functions unique to functional workflow.

- add_global_params(example_argument=d6tflow.IntParameter(default=42))
- resetAll()
- delete(function)
- deleteAll()

7.11 API Docs

- modindex

7.12 Search

- search

Python Module Index

d

Index

W