Data Workspaces Documentation

Release 1.5.0

Max Planck Institute for Software Systems and Data-ken Researcl

Aug 11, 2021

CONTENTS:

CHAPTER

ONE

DATA MANAGEMENT FOR REPRODUCABILITY AND COLLABORATION

Data Workspaces is an open source framework for maintaining the state of a data science project, including data sets, intermediate data, results, and code. It supports reproducability through snapshotting and lineage models and collaboration through a push/pull model inspired by source control systems like Git.

Data Workspaces is installed as a Python 3 package and provides a Git-like command line interface and programming APIs. Specific data science tools and workflows are supported through extensions called *kits*. The goal is to provide the reproducibility and collaboration benefits with minimal changes to your current projects and processes.

Data Workspaces runs on Unix-like systems, including Linux, MacOS, and on Windows via the Windows Subsystem for Linux.

Data Workspaces lets you:

1. Track and version all the different resources for your data science project from one place.
2. Automatically track the full history of your experimental results. Scripts can easily be developed to build reports on these results.
3. Reproduce any prior experiment, including the source data, code, and configuration parameters used.
4. Go back to a prior experiment as a "branching-off" point to explore additional permuations.
5. Collaborate with others on the same project, sharing data, code, and results.
6. Easily reproduce your environment on a new machine to parallelize work.
7. Publish your environment on a site like GitHub or GitLab for others to download and explore.

1.1 1. Introduction

1.1.1 Quick Start

Here is a quick example to give you a flavor of the project, using scikit-learn and the famous digits dataset running in a Jupyter Notebook.

First, install[1] the libary:

```
pip install dataworkspaces
```

Now, we will create a workspace:

[1] See the *Installation section* for more options and details.

```
mkdir quickstart
cd ./quickstart
dws init --create-resources code,results
```

This created our *workspace* (which is a git repository under the covers) and initialized it with two subdirectories, one for the source code, and one for the results. These are special subdirectories, in that they are *resources* which can be tracked and versioned independently.

Now, we are going to add our source data to the workspace. This resides in an external, third-party git repository. It is simple to add:

```
git clone https://github.com/jfischer/sklearn-digits-dataset.git
dws add git --role=source-data --read-only ./sklearn-digits-dataset
```

The first line (`git clone ...`) makes a local copy of the Git repository for the Digits dataset. The second line (`dws add git ...`) adds the repository to the workspace as a resource to be tracked as part of our project. The `--role` option tells Data Workspaces how we will use the resource (as source data), and the `--read-only` option indicates that we should treat the repository as read-only and never try to push it to its `origin`[2] (as you do not have write permissions to the `origin` copy of this repository).

We can see the list of resources in our workspace via the command `dws report status`:

```
$ dws report status
Status for workspace: quickstart
Resources for workspace: quickstart
| Resource                 | Role         | Type               | Parameters                     ␣
↪                                              |
|__________________________|______________|____________________|_______________________________
↪______________________________________________|
| sklearn-digits-dataset | source-data | git                | remote_origin_url=https://
↪github.com/jfischer/sklearn-digits-dataset.git, |
|                          |              |                    | relative_local_path=sklearn-
↪digits-dataset,                                |
|                          |              |                    | branch=master,                 ␣
↪                                              |
|                          |              |                    | read_only=True                 ␣
↪                                              |
| code                     | code         | git-subdirectory | relative_path=code             ␣
↪                                              |
| results                  | results      | git-subdirectory | relative_path=results          ␣
↪                                              |
No resources for the following roles: intermediate-data.
```

Now, we can create a Jupyter notebook for running our experiments:

```
cd ./code
jupyter notebook
```

This will bring up the Jupyter app in your brower. Click on the *New* dropdown (on the right side) and select “Python 3”. Once in the notebook, click on the current title (“Untitled”, at the top, next to “Jupyter”) and change the title to `digits-svc`.

Now, type the following Python code in the first cell:

[2] In Git, each remote copy of a repository is assigned a name. By convention, the `origin` is the copy from which the local copy was cloned.

```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from dataworkspaces.kits.scikit_learn import LineagePredictor, load_dataset_from_resource

# load the data from filesystem into a "Bunch"
dataset = load_dataset_from_resource('sklearn-digits-dataset')

# Instantiate a support vector classifier and wrap it for dws
classifier = LineagePredictor(SVC(gamma=0.001),
                              'multiclass_classification',
                              input_resource=dataset.resource,
                              model_save_file='digits.joblib')

# split the training and test data
X_train, X_test, y_train, y_test = train_test_split(
    dataset.data, dataset.target, test_size=0.5, shuffle=False)

# train and score the classifier
classifier.fit(X_train, y_train)
classifier.score(X_test, y_test)
```

This code is the same as you would write for scikit-learn without dws, except that:

1. we load the dataset from a resource rather than call the lower-level NumPy fuctions (although you can call those if you prefer), and
2. we wrap the support vector classifier instance with a `LineagePredictor`.

It will take a second to train and run the classifier. In the output of the cell, you should then see:

```
Wrote results to results:results.json

0.9688542825361512
```

Now, you can save and shut down your notebook. If you look at the directory `quickstart/results`, you should see a saved model file, `digits.joblib`, and a results file, `results.json`, file with information about your run. We can format and view the results file with the command `dws report results`:

```
$ dws report results
Results file at results:/results.json

General Properties
| Key                    | Value                      |
|________________________|____________________________|
| step                   | digits-svc                 |
| start_time             | 2020-01-14T12:54:00.473892 |
| execution_time_seconds | 0.13                       |
| run_description        | None                       |

Parameters
| Key                     | Value |
|_________________________|_______|
| C                       | 1.0   |
| cache_size              | 200   |
```

(continues on next page)

(continued from previous page)

```
| class_weight            | None  |
| coef0                   | 0.0   |
| decision_function_shape | ovr   |
| degree                  | 3     |
| gamma                   | 0.001 |
| kernel                  | rbf   |
| max_iter                | -1    |
| probability             | False |
| random_state            | None  |
| shrinking               | True  |
| tol                     | 0.001 |
| verbose                 | False |

Metrics
| Key      | Value |
|__________|_______|
| accuracy | 0.969 |

Metrics: classification_report
| Key          | Value                                                                       ␣
↪                                |
|______________|_____________________________________________________________________________
↪________________________________|
| 0.0          | precision: 1.0, recall: 0.9886363636363636, f1-score: 0.
↪9942857142857142, support: 88                     |
| 1.0          | precision: 0.9887640449438202, recall: 0.967032967032967, f1-score: 0.
↪9777777777777779, support: 91   |
| 2.0          | precision: 0.9883720930232558, recall: 0.9883720930232558, f1-score: 0.
↪9883720930232558, support: 86  |
| 3.0          | precision: 0.9753086419753086, recall: 0.8681318681318682, f1-score: 0.
↪9186046511627908, support: 91  |
| 4.0          | precision: 0.9887640449438202, recall: 0.9565217391304348, f1-score: 0.
↪9723756906077348, support: 92  |
| 5.0          | precision: 0.946236559139785, recall: 0.967032967032967, f1-score: 0.
↪9565217391304348, support: 91    |
| 6.0          | precision: 0.989010989010989, recall: 0.989010989010989, f1-score: 0.
↪989010989010989, support: 91     |
| 7.0          | precision: 0.9565217391304348, recall: 0.9887640449438202, f1-score: 0.
↪9723756906077348, support: 89  |
| 8.0          | precision: 0.9361702127659575, recall: 1.0, f1-score: 0.967032967032967,
↪ support: 88                   |
| 9.0          | precision: 0.9278350515463918, recall: 0.9782608695652174, f1-score: 0.
↪9523809523809524, support: 92  |
| micro avg    | precision: 0.9688542825361512, recall: 0.9688542825361512, f1-score: 0.
↪9688542825361512, support: 899 |
| macro avg    | precision: 0.9696983376479764, recall: 0.9691763901507882, f1-score: 0.
↪9688738265020351, support: 899 |
| weighted avg | precision: 0.9696092010839529, recall: 0.9688542825361512, f1-score: 0.
↪9686644837258652, support: 899 |
```

Next, let us take a *snapshot*, which will record the state of the workspace and save the data lineage along with our results:

```
dws snapshot -m "first run with SVC" SVC-1
```

`SVC-1` is the *tag* of our snapshot. If you look in `quickstart/results`, you will see that the results (currently just `results.json`) have been moved to the subdirectory `snapshots/HOSTNAME-SVC-1`, where `HOSTNAME` is the hostname for your local machine). A file, `lineage.json`, containing a full data lineage graph for our experiment has also been created in that directory.

We can see the history of snapshots with the command `dws report history`:

```
$ dws report history

History of snapshots
| Hash     | Tags  | Created             | accuracy | classification_report      | Message␣
↪          |
|_________|_______|_____________________|__________|____________________________|_________
↪___________|
| f1401a8 | SVC-1 | 2020-01-14T13:00:39 |    0.969 | {'0.0': {'precision': 1.. | first␣
↪run with SVC |
1 snapshots total
```

We can also see the lineage for this snapshot with the command `dws report lineage --snapshot SVC-1`:

```
$ dws report lineage --snapshot SVC-1
Lineage for SVC-1
| Resource                | Type        | Details                                  |␣
↪Inputs                             |
|________________________|_____________|__________________________________________|______
↪_________________________________|
| results                 | Step        | digits-svc at 2020-01-14 12:54:00.473892 |␣
↪sklearn-digits-dataset (Hash:635b7182) |
| sklearn-digits-dataset | Source Data | Hash:635b7182                            | None␣
↪                                 |
```

This report shows us that the *results* resource was writen by the *digits-svc* step, which had as its input the resource *sklearn-digits-dataset*. We also know the specific version of this resource (hash 635b71820) and that it is *source data*, not written by another step.

Some things you can do from here:

- Run more experiments and save their results by snapshotting the workspace. If, at some point, we want to go back to our first experiment, we can run: `dws restore SVC-1`. This will restore the state of the source data and code subdirectories, but leave the full history of the results.
- Upload your workspace on GitHub or an any other Git hosting application. This can be to have a backup copy or to share with others. Others can download it via `dws clone`.
- More complex scenarios involving multi-step data pipelines can easily be automated. See the documentation for details.

See the *Tutorial Section* for a continuation of this example.

1.1.2 Installation

Now, let us look into more detail at the options for installation.

Prerequisites

This software runs directly on Linux and MacOSx. Windows is supported by via the Windows Subsystem for Linux. The following software should be pre-installed:

- git
- Python 3.5 or later
- Optionally, the rclone utility, if you are going to be using it to sync with a remote copy of your data.

Installation from the Python Package Index (PyPi)

This is the easiest way to install Data Workspaces is via the Python Package Index at http://pypi.org.

We recommend first creating a virtual environment to contain the Data Workspaces software and any other software needed for your project. Using the standard Python 3 distribution, you can create and *activate* a virtual environment via:

```
python3 -m venv VIRTUAL_ENVIRONMENT_PATH
source VIRTUAL_ENVIRONMENT_PATH/bin/activate
```

If you are using the Anaconda distribution of Python 3, you can create and activate a virtual environment via:

```
conda create --name VIRTUAL_ENVIRONMENT_NAME
conda activate VIRTUAL_ENVIRONMENT_NAME
```

Now that you have your virtual environment set up, we can install the actual library:

```
pip install dataworkspaces
```

To verify that it was installed correctly, run:

```
dws --help
```

Installation via the source tree

You can clone the source tree and install it as follows:

```
git clone git@github.com:data-workspaces/data-workspaces-core.git
cd data-workspaces-python
pip install `pwd`
dws --help # just a sanity check that it was installed correctly
```

1.1.3 Concepts

Data Workspaces provides a thin layer of the Git version control system for easy management of source data, intermediate data, and results for data science projects. A *workspace* is a Git repository with some added metadata to track external resources and experiment history. You can create and manipulate workspaces via `dws`, a command line tool. There is also a programmatic API for integrating more tightly with your data pipeline.

A workspace contains one or more *resources*. Each resource represents a collection of data that has a particular *role* in the project – source data, intermediate data (generated by processing the original source data), code, and results. Resources can be subdirectories in the workspace's Git repository, separate git repositories, local directories, or remote systems (e.g. an S3 bucket or a remote server's files accessed via ssh).

Once the assets of a data science project have been organized into resources, one can do the work of developing the associated software and running experiments. At any point in time, you can take a *snapshot*, which captures the current state of all the resources referenced by the workspace. If you want to go back to a prior state of the workspace or even an individual resource, you can *restore* back to any prior snapshot.

Results resources are handled a little differently than other types: they are always additive. Each snapshot of a results resource takes the current files in the resource and moves it to a snapshot-specific subdirectory. This lets you view and compare the results of all your prior experiments.

You interact with your data workspace through the `dws` command line tool, which like Git, has various subcommands for the actions you might take (e.g. creating a new snapshot, syncing with a remote repository, etc.).

Beyond the basic versioning of your project through snapshots, you can use the *Lineage API* to track each step of your workflow, including inputs/outputs, parameters, and metrics (accuracy, loss, precision, recall, roc, etc.). This lineage data is saved with your snapshots so you can understand how you arrived at each of your results.

1.1.4 Commmand Line Interface

To run the command line interface, you use the `dws` command, which should have been installed into your environment by `pip install`. `dws` operations have the form:

```
dws [GLOBAL_OPTIONS] COMMAND [COMMAND_OPTIONS] [COMMAND_ARGS]
```

Just run `dws --help` for a list of global options and commands.

Commands

Here is a summary of the key commands:

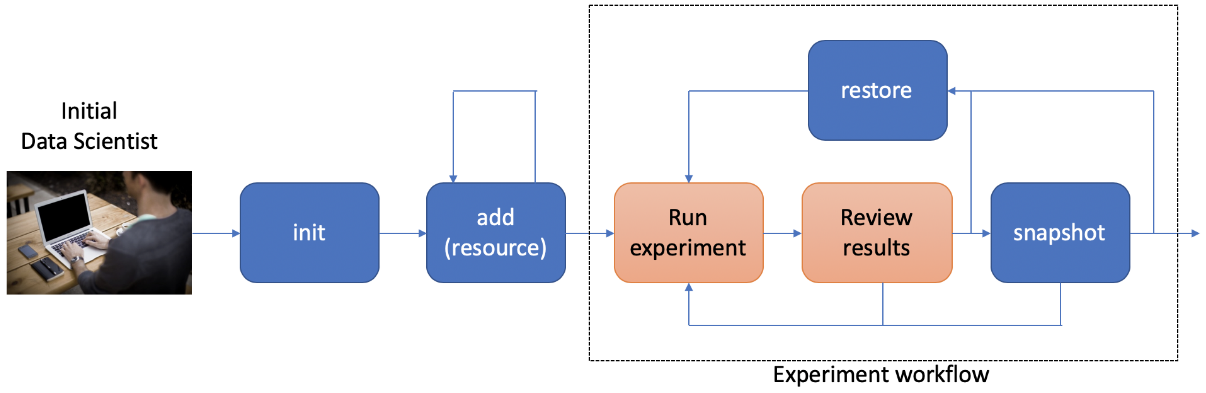
- `init` - initialize a new workspace in the current directory
- `add` - add a *resource* (a git repo, a directory, an s3 bucket, etc.) to the current workspace
- `snapshot` - take a snapshot of the current state of the workspace
- `restore` - restore the state to a prior snapshot
- `publish` - associate a workspace with a remote git repository (e.g. on GitHub)
- `push` - push a workspace and all resources to their (remote) origins
- `pull` - pull the workspace and all resources from their (remote) origins
- `clone` - clone a workspace and all the associated resources to the local machine
- `report` - various reports about the workspace

- `run` - run a command and capture the lineage. This information is saved in a file for future calls to the same command. *(not yet implemented)*

See the *Command Reference* section for a full description of all commands and their options.

Workflow

To put these commands in context, here is a typical workflow for the initial data scientist on a project:

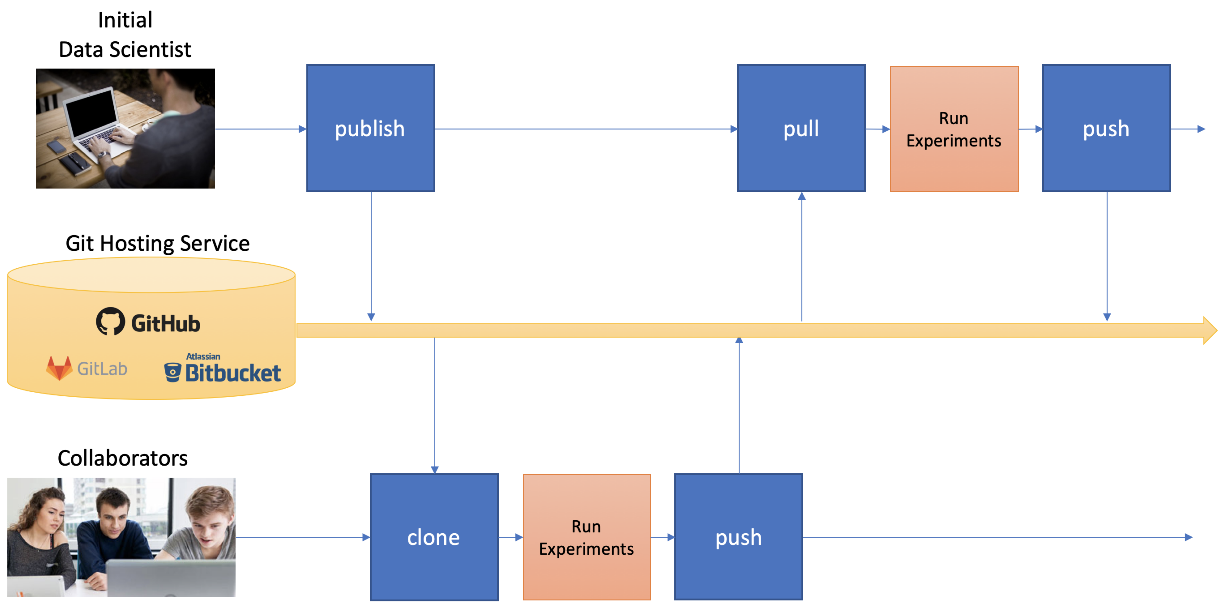


The person starting the project creates a new workspace on their local machine using the `init` command. Next, they need to tell the data workspace about their code, data sets, and places where they will store intermediate data and results. If subdirectories of the main workspace are sufficient, they can do this as a part of the `init` command, using the `--create-resources` option. Otherwise, they use the add command to define each *resource* associated with their project.

The data scientist can now run their experiements. This is typically an iterative process, represented in the picture by the dashed box labeled "Experiment Workflow". Once they have finished a complete experiment, then can use the `snapshot` command to capture the state of their workspace. They can go back and run further experiments, taking a snapshot each time they have something interesting. They can also go back to a prior state using the `restore` command.

Collaboration

At some point, the data scientist will want to copy their project to a remote service for sharing (and backup). Data Workspaces can use any Git hosting service for this (e.g. GitHub, GitLab, or BitBucket) and does not need any special setup. Here is an overview of collaborations facilitated by Data Workspaces:

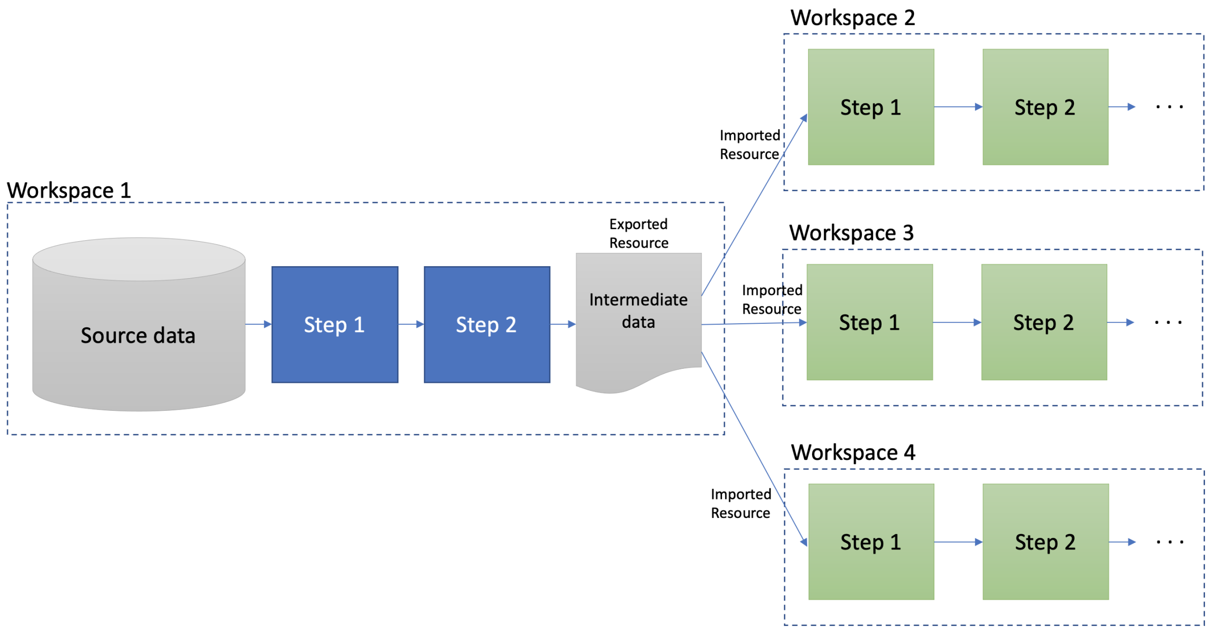


First, the data scientist creates an empty git repository on the remote `origin` (e.g. GitHub, GitLab, or BitBucket) and then runs the `publish` command to associate the `origin` with the workspace and update the `origin` with the full history of the workspace.

A new collaborator can use the `clone` command to copy the workspace down to their local machine. They can then run experiments and take snapshots, just like the original data scientisst. When readly, then can upload their changes to the via the `push` command. Others can then use the `pull` command to download these changes to their workspace. This process can be repeated as many times as necessary, and multiple collaborators can overlap their work.

1.1.5 Sharing lineage across workspaces

In some more complex scenarios, it makes sense to split a data pipeline across multiple workspaces. For example, an intial sequence of steps may generate data used across multiple downstream pipelines. As shown in the picture below, the initial pipeline can be its own self contained workspace. The intermediate data to be used downstream is then an *exported* resource. Such resources save their lineage in the resource itself, in a file named `lineage.json`.



The downstream pipelines can then be separate workspaces that *import* the resource. This causes the lineage graph of the imported resource to be included in the lineage graph of the importing workspace. Thus, end to end lineage is still captured, even across workspaces.

For more details, see the *command reference*, specifically the `-export` and `--import` options of the `dws add [resource_type]` subcommands.

1.2 2. Tutorial

Let's build on the *Quick Start*. If you haven't already, run through it so that you have a `quickstart` workspace with one tag (`SVC-1`).

1.2.1 Further Experiments

Now, let's try to use Logistic Regression. Create a new Jupyter notebook called `digits-lr` in the `code` subdirectory of `quickstart`. Enter the following code into a notebook cell:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from dataworkspaces.kits.scikit_learn import LineagePredictor, load_dataset_from_resource

# load the data from filesystem into a "Bunch"
dataset = load_dataset_from_resource('sklearn-digits-dataset')

# split the training and test data
X_train, X_test, y_train, y_test = train_test_split(
      dataset.data, dataset.target, test_size=0.5, shuffle=False)

# Run the a grid search to find the best parameters
```

(continues on next page)

(continued from previous page)

```
gs_params={'C':[1e-3, 1e-2, 1e-1, 1, 1e2], 'solver':['lbfgs'],
           'multi_class':['multinomial']}
cv = GridSearchCV(LogisticRegression(), gs_params, cv=5, scoring='accuracy')
cv.fit(X_train, y_train)

# Instantiate a LogisticRegression classifier with the best parameters
# and wrap it for dws
classifier = LineagePredictor(LogisticRegression(**cv.best_params_),
                              'multiclass_classification',
                              input_resource=dataset.resource,
                              model_save_file='digits.joblib')


# train and score the classifier
classifier.fit(X_train, y_train)
classifier.score(X_test, y_test)
```

There are two differences from our previous notebook:

1. we use a LogisticRegression classifier rather than a Support Vector classifier, and
2. Before calling our wrapped classifier, we run a grid search to find the best combination of model parameters.

If you run this cell, you should see several no-convergence warnings (some of the values for `C` must be bad for this data set) and then a final accuracy result, around 94%

Ok, so our Logistic Regression accuracy of 0.94 is not as good as we obtained from the Support Vector Classifier (0.97). Let's take a snapshot anyway, so we have this experiment for future reference. Maybe someone will ask us, "Did you try Logistic Regession?", and we can show them the full results and even use a `restore` command to re-run the experiment for them. Here's how to take the snapshot:

```
dws snapshot -m "Logistic Regession experiment" LR-1
```

We can see both snapshots with the command `dws report history`:

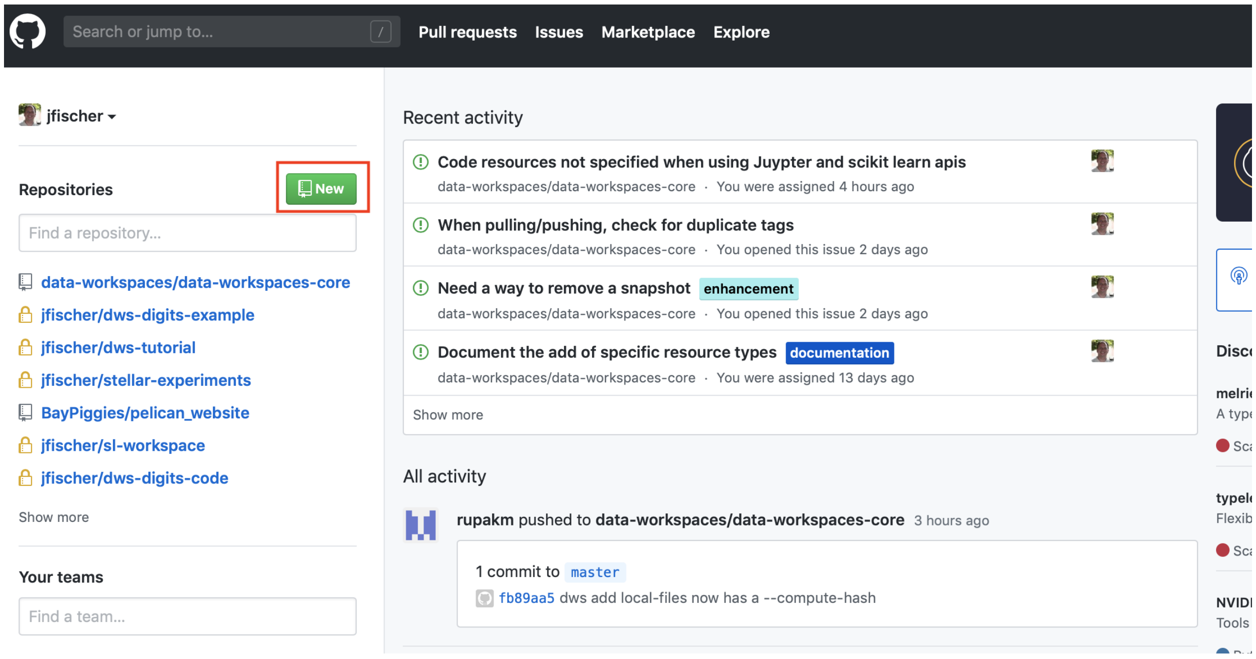
```
$ dws report history

History of snapshots
| Hash    | Tags  | Created                | accuracy | classification_report      | Message␣
↪                         |
|_________|_______|_______________________|__________|___________________________|_________
↪_______________________|
| bf9fb37 | LR-1  | 2020-01-14T14:27:37 |     0.94 | {'0.0': {'precision': 0.. |␣
↪Logistic Regession experiment |
| f1401a8 | SVC-1 | 2020-01-14T13:00:39 |    0.969 | {'0.0': {'precision': 1.. | first␣
↪run with SVC             |
2 snapshots total
```

1.2.2 Publishing a workspace

Now, we will publish our workspace on GitHub. A similar approach can be taken for other code hosting services like BitBucket or GitLab.

The first few steps are GitHub-specific, but the `dws` commands will work across all hosting services.

First, create an account on GitHub if you do not already have one. Next, go to your front page on GitHub and click on the green new repository button on the left side of the page:



data-workspaces/data-workspaces-core
jfischer/dws-digits-example
jfischer/dws-tutorial
jfischer/stellar-experiments
BayPiggies/pelican_website
jfischer/sl-workspace
jfischer/dws-digits-code

Code resources not specified when using Juypter and scikit learn apis
data-workspaces/data-workspaces-core · You were assigned 4 hours ago

When pulling/pushing, check for duplicate tags
data-workspaces/data-workspaces-core · You opened this issue 2 days ago

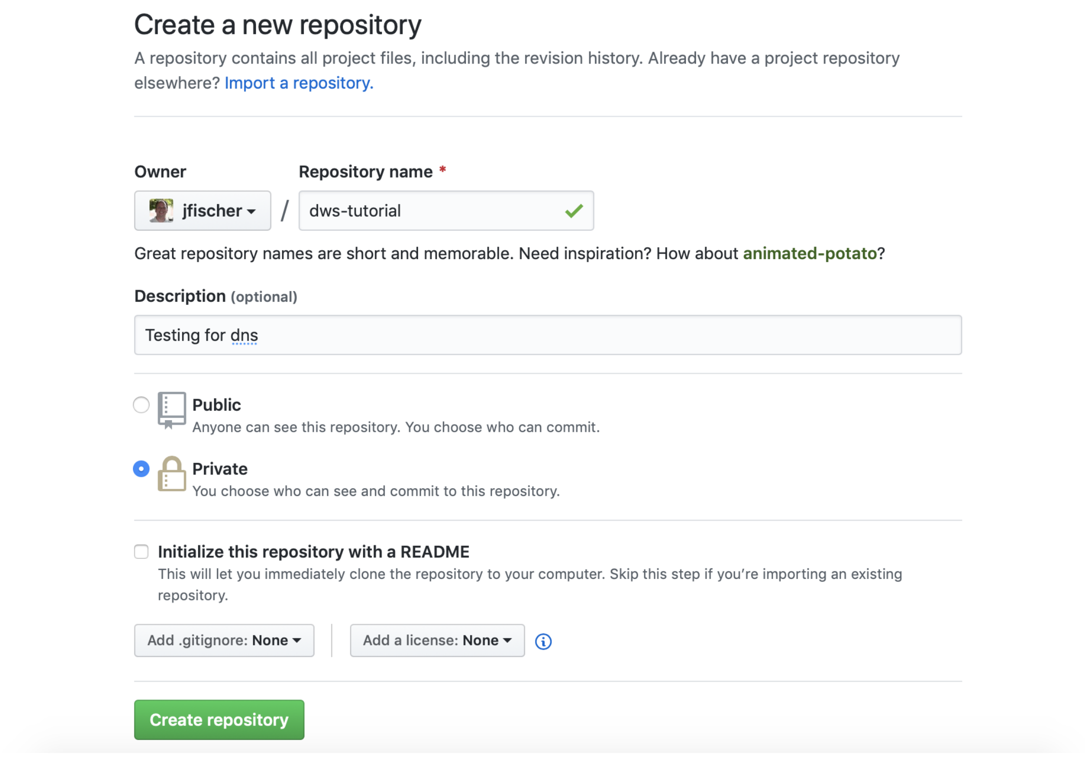
Need a way to remove a snapshot enhancement
data-workspaces/data-workspaces-core · You opened this issue 2 days ago

Document the add of specific resource types documentation
data-workspaces/data-workspaces-core · You were assigned 13 days ago

rupakm pushed to data-workspaces/data-workspaces-core 3 hours ago

1 commit to `master`
`fb89aa5` dws add local-files now has a --compute-hash

You should now get a dialog like this:


Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner: jfischer / Repository name *: dws-tutorial

Great repository names are short and memorable. Need inspiration? How about animated-potato?

Description (optional): Testing for dns

( ) Public — Anyone can see this repository. You choose who can commit.

(● Private) — You choose who can see and commit to this repository.

☐ Initialize this repository with a README — This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

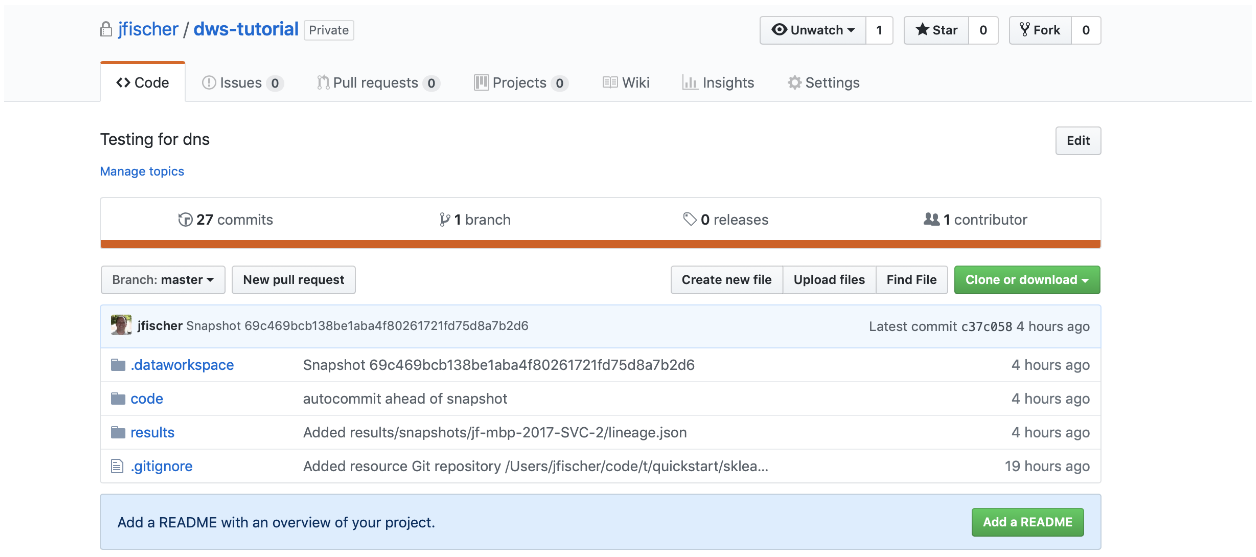
Fill in a name for your repository (in this case, `dws-tutorial`) and select whether you want it to be public (visible to the work) or private (only visible to those you explicitly grant access). You won't need a README file, .gitignore, or license file, as we will be initializing the repository from your local copy. Go ahead and click on the "Create Repository" button.

Now, back on the command line, go to the directory containing the `quickstart` workspace on your local machine. Run the following command replacing `YOUR_USERNAME` with your GitHub username:

```
dws publish git@github.com:YOUR_USERNAME/dws-tutorial.git
```

You have published your workspace and its history to a GitHub Repository.

At this point, if you refresh the page for this repository on GitHub, you should see something like this:



Testing for dns

27 commits | 1 branch | 0 releases | 1 contributor

jfischer Snapshot 69c469bcb138be1aba4f80261721fd75d8a7b2d6 — Latest commit c37c058 4 hours ago

.dataworkspace	Snapshot 69c469bcb138be1aba4f80261721fd75d8a7b2d6	4 hours ago
code	autocommit ahead of snapshot	4 hours ago
results	Added results/snapshots/jf-mbp-2017-SVC-2/lineage.json	4 hours ago
.gitignore	Added resource Git repository /Users/jfischer/code/t/quickstart/sklea...	19 hours ago

Add a README with an overview of your project.

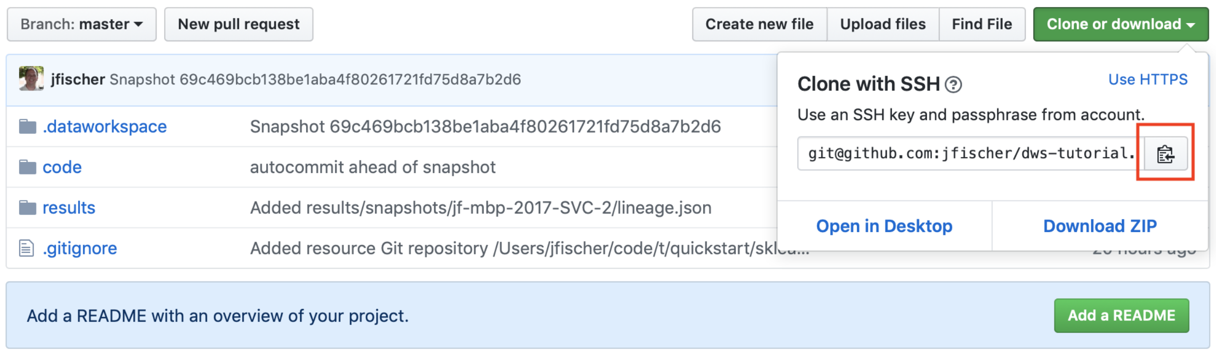
You have successfully published your workspace!

1.2.3 Cloning a workspace

Now, we want to use this workspace on a new machine (perhaps your own or perhaps belonging to a collaborator). First, make certain that the account on the second machine has at least read access to the repository. If you will be pushing updates from this account, it will also need write access to the repo. Next, make sure that your software dependencies are installed (e.g. Jupyter, NumPy, and Scikit-learn) and then install the Data Workspaces library into your local environment:

```
pip install dataworkspaces
```

From a browser on your second machine, go back to the GitHub page for your repository and click on the "Clone or download" button. It should show you a URL for cloning via SSH. Click on the clipboard icon to the right of the URL to copy the URL to your machine's clipboard:



jfischer Snapshot 69c469bcb138be1aba4f80261721fd75d8a7b2d6

Clone with SSH

Use an SSH key and passphrase from account.

```
git@github.com:jfischer/dws-tutorial.
```

.dataworkspace	Snapshot 69c469bcb138be1aba4f80261721fd75d8a7b2d6
code	autocommit ahead of snapshot
results	Added results/snapshots/jf-mbp-2017-SVC-2/lineage.json
.gitignore	Added resource Git repository /Users/jfischer/code/t/quickstart/sklea...

Add a README with an overview of your project.

Then, on your second machine, go to the directory you intend to be the parent of th workspace (in this case `~/workspaces`) and run the following:

```
dws clone GITHUB_CLONE_URL
```

where `GITHUB_CLONE_URL` is the URL you copied to your clipboard.

It should ask you for the hostname you want to use to identify this machine. It defaults to the system hostname.

By default, the clone will be in the directory `./quickstart`, since "quickstart" was the name of the original repo. You can change this by adding the desired local directory name to the command line.

We can now change to the workspace's directory and run the history command:

```
$ cd ./quickstart
$ dws report history
History of snapshots
| Hash    | Tags  | Created             | accuracy | classification_report      | Message␣
↪                      |
|_________|_______|_____________________|__________|____________________________|_________
↪______________________|
| bf9fb37 | LR-1  | 2020-01-14T14:27:37 |     0.94 | {'0.0': {'precision': 0.. |␣
↪Logistic Regession experiment |
| f1401a8 | SVC-1 | 2020-01-14T13:00:39 |    0.969 | {'0.0': {'precision': 1.. | first␣
↪run with SVC                |
2 snapshots total
```

We see the full history from the original workspace!

1.2.4 Sharing updates

Let's re-run the Support Vector classifier evaluation on the second machine and see if we reproduce our results. First, go to the `code` subdirectory in your workspace. Start the Jupyter notebook as follows:

```
jupyter notebook digits-svc.ipynb
```

This should bring up a browser with the notebook. You should see the code from our first experiment. Run the cell. You should get close the same results as on the first machine (0.97 accuracy). Save and shutdown the notebook.

Now, take a snapshot:

```
dws snapshot -m "reproduce on second machine" SVC-2
```

We have tagged this snapshot with the tag `SVC-2`. We want to push the entire workspace to GitHub. This can be done as follows:

```
dws push
```

After the push, the `origin` respository on GitHub has been updated with the latest snapshot and results. We can now go back to the origin machine where we created the workspace, and download the changes. To do so, start up a command line window, go into the workspace's directory on the first machine, and run:

```
dws pull
```

After the pull, we should see the experiment we ran on the second machine:

```
$ dws report history
History of snapshots
| Hash    | Tags  | Created             | accuracy | classification_report      | Message␣
↪                      |
|_________|_______|_____________________|__________|____________________________|_________
↪______________________|
```

(continues on next page)

(continued from previous page)

```
| 2c195ba | SVC-2 | 2020-01-14T15:20:23 |    0.969 | {'0.0': {'precision': 1.. |␣
↪reproduce on second machine   |
| bf9fb37 | LR-1  | 2020-01-14T14:27:37 |     0.94 | {'0.0': {'precision': 0.. |␣
↪Logistic Regession experiment |
| f1401a8 | SVC-1 | 2020-01-14T13:00:39 |    0.969 | {'0.0': {'precision': 1.. | first␣
↪run with SVC             |
3 snapshots total
```

1.3 3. Command Reference

In this section, we describe the full command line interface for Data Workspaces. This interface is built around a script `dws`, which is installed into your path when you install the `dataworkspaces` package. The overall interface for `dws` is:

```
dws [--batch] [--verbose] [--help] COMMAND [--help] [OPTIONS] [ARGS]...
```

`dws` has three options common to all commands:

- `--batch`, which runs the command in a mode that never asks for user confirmation and will error out if it absolutely requires an input (useful for automation),
- `--verbose`, which will print a lot of detail about what will be and has been done for a command (useful for debugging), and
- `--help`, which prints these common options and a list of available commands.

Next on your command line comes the command name (e.g. `init`, `clone`, `snapshot`). Each command has its own arguments and options, as documented below. All commands take a `--help` argument, which will print the specific options and arguments for the command. Finally, the `add` subcommand has further subcommands, representing the individual resource types (e.g. *git*, *local-files*, *rclone*).

1.4 4. Lineage API

The Lineage API is provided by the module `dataworkspaces.lineage`.

This module provides an API for tracking *data lineage* – the history of how a given result was created, including the versions of original source data and the various steps run in the *data pipeline* to produce the final result.

The basic idea is that your workflow is a sequence of *pipeline steps*:

```
----------     ----------     ----------     ----------
|        |     |        |     |        |     |        |
| Step 1 |---->| Step 2 |---->| Step 3 |---->| Step 4 |
|        |     |        |     |        |     |        |
----------     ----------     ----------     ----------
```

A step could be a command line script, a Jupyter notebook or perhaps a step in an automated workflow tool (e.g. Apache Airflow). Each step takes a number of *inputs* and *parameters* and generates *outputs*. The inputs are resources in your workspace (or subpaths within a resource) from which the step will read to perform its task. The parameters are configuration values passed to the step (e.g. the command line arguments of a script). The outputs are the resources (or subpaths within a resource), which are written to by the step. The outputs may represent results or intermediate data to be consumed by downstream steps.

The lineage API captures this data for each step. Here is a view of the data captured:

```
                   Parameters
                    ||  ||  ||
                    \/  \/  \/
                  ------------
                =>|          |=>
Input resources =>|  Step i  |=> Output resources
                =>|          |=>
                  ------------
                       /\
                       ||
                      Code
                  Dependencies
```

To do this, we need use the following classes:

- *ResourceRef* - A reference to a resource for use as a step input or output. A ResourceRef contains a resource name and an optional path within that resource. This lets you manage lineage down to the directory or even file level. The APIs also support specifying a path on the local filesystem instead of a ResourceRef. This path is automatically resolved to a ResourceRef (it must map to the a location under the local path of a resource). By storing :class:`~ResourceRef`s instead of hard-coded filesystem paths, we can include non-local resources (like an S3 bucket) and ensure that the workspace is easily deployed on a new machine.
- *Lineage* - The main lineage object, instantiated at the start of your step. At the beginning of your step, you specify the inputs, parameters, and outputs. At the end of the step, the data is saved, along with any results you might have from that step. Lineage instances are context managers, which means you can use a `with` statement to manage their lifecycle.
- *LineageBuilder* - This is a helper class to guide the creation of your lineage object.

Example

Here is an example usage of the lineage API in a command line script:

```python
import argparse
from dataworkspaces.lineage import LineageBuilder

def main():
    ...
    parser = argparse.ArgumentParser()
    parser.add_argument('--gamma', type=float, default=0.01,
                        help="Regularization parameter")
    parser.add_argument('input_data_dir', metavar='INPUT_DATA_DIR', type=str,
                        help='Path to input data')
    parser.add_argument('results_dir', metavar='RESULTS_DIR', type=str,
                        help='Path to where results should be stored')
    args = parser.parse_args()
    ...
    # Create a LineageBuilder instance to specify the details of the step
    # to the lineage API.
    builder = LineageBuilder()\
                .as_script_step()\
                .with_parameters({'gamma':args.gamma})\
                .with_input_path(args.input_data_dir)\
                .as_results_step(args.results_dir)
```

(continues on next page)

(continued from previous page)

```
    # builder.eval() will construct the lineage object. We call it within a
    # with statement to get automatic save/cleanup when we leave the
    # with block.
    with builder.eval() as lineage:

        ... do your work here ...

        # all done, write the results
        lineage.write_results({'accuracy':accuracy,
                               'precision':precision,
                               'recall':recall,
                               'roc_auc':roc_auc})

    # When leaving the with block, the lineage is automatically saved to the
    # workspace. If an exception is thrown, the lineage is not saved, but the
    # outputs are marked as being in an unknown state.

    return 0

# boilerplate to call our main function if this is called as a script.
if __name__ == '__main__':
    sys.exit(main())
```

1.4.1 Classes

class dataworkspaces.lineage.**ResourceRef**(*name: str*, *subpath: Optional[str] = None*)

A namedtuple that is used to identify an input or output of a step. The `name` parameter is the name of a resource. The optional `subpath` parameter is a relative path within that resource. The subpath lets you store inputs/outputs from multiple steps within the same resource and track them independently.

class dataworkspaces.lineage.**Lineage**

This is the main object for tracking the execution of a step. Rather than instantiating it directly, use the *LineageBuilder* class to construct your *Lineage* instance.

abort()

The step has failed, so we mark its outputs in an unknown state. If you create the lineage via a "with" statement, then this will be called for you automatically.

add_output_path(*path: str*) → None

Resolve the path to a resource name and subpath. Add that to the lineage as an output of the step. From this point on, if the step fails (*abort()* is called), the associated resource and subpath will be marked as being in an "unknown" state.

add_output_ref(*ref:* dataworkspaces.utils.lineage_utils.ResourceRef)

Add the resource reference to the lineage as an output of the step. From this point on, if the step fails (*abort()* is called), the associated resource and subpath will be marked as being in an "unknown" state.

add_param(*name: str*, *value*) → None

Add or update one of the step's parameters.

complete()

The step has completed. Save the outputs. If you create the lineage via a "with" statement, then this will be called for you automatically.

`class dataworkspaces.lineage.`**`ResultsLineage`**

Lineage for a results step. This subclass is returned by the `LineageBuilder` when `as_results_step()` is called. This marks the `Lineage` object as generating results. It adds the `write_results()` method for writing a JSON summary of the final results.

Results resources will also have a `lineage.json` file added when the next snapshot is taken. This file contains the full lineage graph collected for the resource.

`write_results`(*metrics: Dict[str, Any]*)

Write a `results.json` file to the results directory specified when creating the lineage object (e.g. via `as_results_step()`). This json file contains information about the step execution (e.g. start time), parameters, and the provided metrics.

`class dataworkspaces.lineage.`**`LineageBuilder`**

Use this class to declaratively build `Lineage` objects. Instantiate a LineageBuilder instance, and call a sequence of configuration methods to specify your inputs, parameters, your workspace (if the script is not already inside the workspace), and whether this is a results step. Each configuration method returns the builder, so you can chain them together. Finally, call `eval()` to instantiate the `Lineage` object.

Configuration Methods

To specify the workflow step's name, call one of:

- `as_script_step()` - the script's name will be used to infer the step and the associated code resource
- with_step_name - explicitly specify the step name

To specify the parameters of the step (e.g. command line arguments), use the `with_parameters()` method.

To specify the input of the step call one or more of:

- `with_input_path()` - resolve the local filesystem path to a resource and subpath and add it to the lineage as inputs. May be called more than once.
- `with_input_paths()` - resolve a list of local filesystem paths to resources and subpaths and add them to the lineage as inputs. May be called more than once.
- `with_input_ref()` - add the resource and subpath to the lineage as an input. May be called more than once.
- `with_no_inputs()` - mutually exclusive with the other input methods. This signals that there are no inputs to this step.

To specify code resource dependencies for the step, you can call `with_code_ref()`. For command-line Python scripts, the main code resource is handled automatically in `as_script_step()`. Other subclasses of the LineageBuilder may provide similar functionality (e.g. the LineageBuilder for JupyterNotebooks will try to figure out the resource containing your notebook and set it in the lineage).

If you need to specify the workspace's root directory, use the `with_workspace_directory()` method. Otherwise, the lineage API will attempt to infer the workspace directory by looking at the path of the script.

Call `as_results_step()` to indicate that this step is producing results. This will add a method `write_results()` to the `Lineage` object returned by `eval()`. The method `as_results_step()` takes two parameters: *results_dir* and, optionally, *run_description*. The results directory should correspond to either the root directory of a results resource or a subdirectory within the resource. If you have multiple steps of your workflow that produce results, you can create separate subdirectories for each results-producing step.

Example

Here is an example where we build a `Lineage` object for a script, that has one input, and that produces results:

```
lineage = LineageBuilder()\
            .as_script_step()\
            .with_parameters({'gamma':0.001})\
            .with_input_path(args.intermediate_data)\
            .as_results_step('../results').eval()
```

Methods

as_results_step(*results_dir: str*, *run_description: Optional[str] = None*) → *dataworkspaces.lineage.LineageBuilder*

as_script_step() → *dataworkspaces.lineage.LineageBuilder*

eval() → *dataworkspaces.lineage.Lineage*

Validate the current configuration, making sure all required properties have been specified, and return a `Lineage` object with the requested configuration.

with_code_path(*path: str*) → *dataworkspaces.lineage.LineageBuilder*

with_code_ref(*ref:* dataworkspaces.utils.lineage_utils.ResourceRef) → *dataworkspaces.lineage.LineageBuilder*

with_input_path(*path: str*) → *dataworkspaces.lineage.LineageBuilder*

with_input_paths(*paths: List[str]*) → *dataworkspaces.lineage.LineageBuilder*

with_input_ref(*ref:* dataworkspaces.utils.lineage_utils.ResourceRef) → *dataworkspaces.lineage.LineageBuilder*

with_no_inputs() → *dataworkspaces.lineage.LineageBuilder*

with_parameters(*parameters: Dict[str, Any]*) → *dataworkspaces.lineage.LineageBuilder*

with_step_name(*step_name: str*) → *dataworkspaces.lineage.LineageBuilder*

with_workspace_directory(*workspace_dir: str*) → *dataworkspaces.lineage.LineageBuilder*

1.4.2 Using Lineage

Once you have instrumented the individual steps of your workflow, you can run the steps as normal. Lineage data is stored in the directory `.dataworkspace/current_lineage`, but not checked into the associated Git repository.

When you take a snapshot, this lineage data is copied to `.dataworkspace/snapshot_lineage/HASH`, where HASH is the hashcode associated with the snapshot, and checked into git. This data is available as a record of how you obtained the results associated with the snapshot. In the future, more tools will be provided to analyze and operate on this lineage (e.g. replaying workflows).

When you restore a snapshot, the lineage data assocociated with the snapshot is restored to `.dataworkspace/current_lineage`.

Consistency

In order to fully track the status of your workflow, we make a few restrictions:

1. Independent steps should not overwrite the same ResourceRef or a ResourceRef where one ResourceRef refers to the subdirectory of another ResourceRef.
2. A step's execution should not transitively depend on two different versions of the same ResourceRef. If you try to run a step in this situation, an exception will be thrown.

These restrictions should not impact reasonable workflows in practice. Furthermore, they help to catch some common classes of errors (e.g. not rerunning all the dependent steps when a change is made to an input).

1.5 5. Kits Reference

In this section, we cover *kits*, integrations with various data science libraries and infrastructure provided by Data Workspaces.

1.5.1 Jupyter

Integration with Jupyter notebooks. This module provides a `LineageBuilder` subclass to simplify Lineage for Notebooks.

It also provides a collection of IPython *magics* (macros) for working in Jupyter notebooks.

class `dataworkspaces.kits.jupyter.`**`NotebookLineageBuilder`**(*results_dir: str*, *step_name: Optional[str] = None*, *run_description: Optional[str] = None*)

Notebooks are the final step in a pipeline (and potentially the only step). We customize the standard lineage builder to get the step name from the notebook's name and to always have a results directory.

If you are not running this notebook in a server context (e.g. via nbconvert), the step name won't be available. In that case, you can explicitly pass in the step name to the constructor.

`dataworkspaces.kits.jupyter.`**`get_step_name_for_notebook`**() → Optional[str]

Get the step name for a notebook by getting the path and then extracting the base name. In some situations (e.g. running on the command line via nbconvert), the notebook name is not available. We return None in those cases.

`dataworkspaces.kits.jupyter.`**`is_notebook`**() → bool

Return true if this code is running in a notebook.

Magics

This module also provides a collection of IPython magics (macros) to simplify interactions with your data workspace when develping in a Jupyter Notebook.

Limitations

Currently these magics are only supported in interactive Jupyter Notebooks. They do not run properly within JupyterLab (we are currently working on an extension specific to JupyterLab), the `nbconvert` command, or if you run the entire notebook with "Run All Cells".

To develop a notebook interactively using the DWS magic commands and then run the same notebook in batch mode, you can set the variable `DWS_MAGIC_DISABLE` in your notebook, ahead of the call to load the magics (`%load_ext`). If you set it to `True`, the commands will be loaded in a disabled state and will run with no effect. Setting `DWS_MAGIC_DISABLE` to `False` will load the magics in the enabled state and run all commands normally.

Loading the magics

To load the magics, run the following in an interactive cell of your Jupyter Notebook:

```
import dataworkspaces.kits.jupyter
%load_ext dataworkspaces.kits.jupyter
```

If the load runs correctly, you should see output like this in your cell:

Ran DWS initialization. The following magic commands have been added to your notebook:

- `%dws_info` - *print information about your dws environment*
- `%dws_history` - *print a history of snapshots in this workspace*
- `%dws_snapshot` - *save and create a new snapshot*
- `%dws_lineage_table` - *show a table of lineage for the workspace resources*
- `%dws_lineage_graph` - *show a graph of lineage for a resource*
- `%dws_results` - *show results from a run (results.json file)*

Run any command with the `--help` *option to see a list of options for that command. The variable* `DWS_JUPYTER_NOTEBOOK` *has been added to your variables, for use in future DWS calls.*

If you want to disable the DWS magic commands (e.g. when running in a batch context), set the variable `DWS_MAGIC_DISABLE` *to* `True` *ahead of the* `%load_ext` *call.*

Magic Command reference

We now describe the command options for the individual magics.

%dws_info

usage: dws_info [-h]

Print some information about this workspace

optional arguments:

-h, --help	show this help message and exit

%dws_history

usage:
dws_history [-h] [–max-count MAX_COUNT] [–tail] [–baseline TAG_OR_HASH]

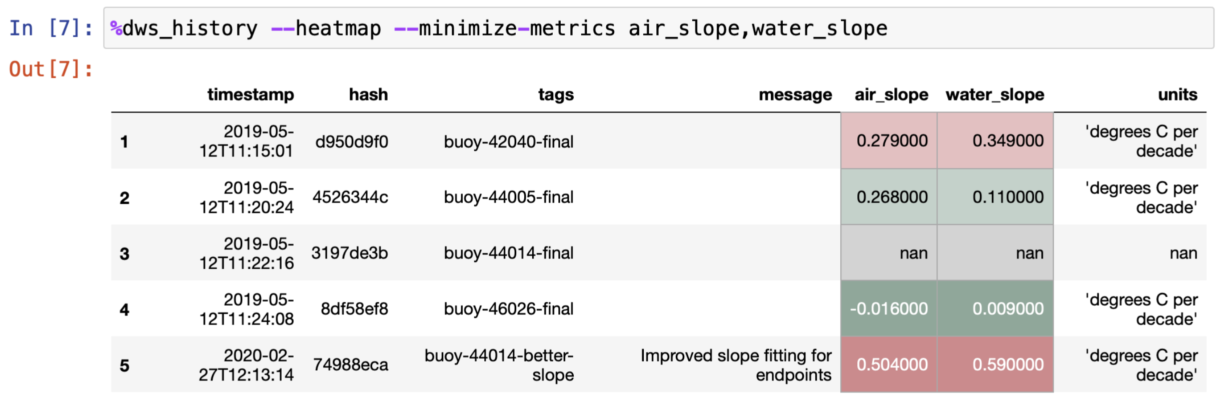
[–heatmap] [maximize-metrics METRICS] [–minimize-metrics METRICS]

Print a history of snapshots in this workspace

optional arguments:

-h, --help	show this help message and exit
--max-count MAX_COUNT	Maximum number of snapshots to show
--tail	Just show the last 10 entries in reverse order
--baseline TAG_OR_HASH	Snapshot tag or hash to use as a basis for metrics comparison.
--heatmap	Show a heatmap for metrics columns
--maximize-metrics METRICS	Metrics where larger values are better (e.g. accuracy)
--minimize-metrics METRICS	Metrics where smaller values are better (e.g. loss)

For easy visualization of results, the `%dws_history` command supports two styles of color coding. The `--heatmap` option will color code the background of metrics cells from dark red (worst results) to dark green (best results). For common metrics, like accuracy and loss, DWS knows the directionality of the metric (higher is better vs. lower is better). For less common or custom metrics, you can use the `--maximize-metrics` and `--minimize-metrics` options to specify this. Here is an example heatmap:



```
In [7]: %dws_history --heatmap --minimize-metrics air_slope,water_slope
Out[7]:
```

	timestamp	hash	tags	message	air_slope	water_slope	units
1	2019-05-12T11:15:01	d950d9f0	buoy-42040-final		0.279000	0.349000	'degrees C per decade'
2	2019-05-12T11:20:24	4526344c	buoy-44005-final		0.268000	0.110000	'degrees C per decade'
3	2019-05-12T11:22:16	3197de3b	buoy-44014-final		nan	nan	nan
4	2019-05-12T11:24:08	8df58ef8	buoy-46026-final		-0.016000	0.009000	'degrees C per decade'
5	2020-02-27T12:13:14	74988eca	buoy-44014-better-slope	Improved slope fitting for endpoints	0.504000	0.590000	'degrees C per decade'

The second style of coloring takes a baseline snapshot. Any metric values better than the baseline have their text colored green, any metric values close to the baseline are bold black text, and any metric values worse than the baseline are colored red. The `--baseline=SNAPSHOT` option enables this display mode. Here is an example:

```
In [8]: %dws_history --baseline=buoy-42040-final --minimize-metrics air_slope,water_slope
```

Out[8]:

	timestamp	hash	tags	message	air_slope	water_slope	units
1	2019-05-12T11:15:01	d950d9f0	buoy-42040-final		**0.279000**	**0.349000**	'degrees C per decade'
2	2019-05-12T11:20:24	4526344c	buoy-44005-final		0.268000	0.110000	'degrees C per decade'
3	2019-05-12T11:22:16	3197de3b	buoy-44014-final		nan	nan	nan
4	2019-05-12T11:24:08	8df58ef8	buoy-46026-final		-0.016000	0.009000	'degrees C per decade'
5	2020-02-27T12:13:14	74988eca	buoy-44014-better-slope	Improved slope fitting for endpoints	0.504000	0.590000	'degrees C per decade'

%dws_snapshot

usage: dws_snapshot [-h] [-m MESSAGE] [-t TAG]

Save the notebook and create a new snapshot

optional arguments:

-h, --help	show this help message and exit
-m MESSAGE, --message MESSAGE	Message describing the snapshot
-t TAG, --tag TAG	Tag for the snapshot. Note that a given tag can only be used once (without deleting the old one).

%dws_lineage_table

usage: dws_lineage_table [-h] [–snapshot SNAPSHOT]

Show a table of lineage for the workspace's resources

optional arguments:

-h, --help	show this help message and exit
--snapshot SNAPSHOT	If specified, print lineage as of the specified snapshot hash or tag

%dws_lineage_graph

usage: dws_lineage_table [-h] [–resource RESOURCE] [–snapshot SNAPSHOT]

Show a graph of lineage for a resource

optional arguments:

-h, --help	show this help message and exit
--resource RESOURCE	Graph lineage from this resource. Defaults to the results resource. Error if not specified and there is more than one.
--snapshot SNAPSHOT	If specified, graph lineage as of the specified snapshot hash or tag

%dws_results

usage: dws_results [-h] [–resource RESOURCE] [–snapshot SNAPSHOT]

Show results from a run (results.json file)

optional arguments:

-h, --help show this help message and exit

--resource RESOURCE Look for the results.json file in this resource. Otherwise, will look in all results resources and return the first match.

--snapshot SNAPSHOT If specified, get results as of the specified snapshot or tag. Otherwise, looks at current workspace and then most recent snapshot.

1.5.2 Scikit-learn

This module (`dataworkspaces.kits.scikit_learn`) provides integration with the scikit-learn framework. The main class provided here is *`LineagePredictor`*, which wraps any class following sklearn's predictor protocol. It captures inputs, model parameters and results. This module also provides *`Metrics`* and its subclasses, which support the computation of common metrics and the writing of them to a results file. Finally, there is `train_and_predict_with_cv()`, which runs a common sklearn classification workflow, including grid search.

`class dataworkspaces.kits.scikit_learn.`**`BinaryClassificationMetrics`**`(`*expected*, *predicted*, *sample_weight=None*`)`

Given an array of expected (target) values and the actual predicted values from a classifier, compute metrics that make sense for a binary classifier, including accuracy, precision, recall, roc auc, and f1 score.

`print_metrics``(`*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*`)` → None

Print the metrics to a file

`score``()` → float

Metric for binary classification is accuracy

`class dataworkspaces.kits.scikit_learn.`**`LineagePredictor`**`(`*predictor*, *metrics: Union[str, type]*, *input_resource: Union[str, dataworkspaces.utils.lineage_utils.ResourceRef]*, *results_resource: Optional[Union[str, dataworkspaces.utils.lineage_utils.ResourceRef]] = None*, *model_save_file: Optional[str] = None*, *workspace_dir: Optional[str] = None*, *verbose: bool = False*`)`

This is a wrapper for adding lineage to any predictor in sklearn. To use it, instantiate the predictor (for classification or regression) and then create a new instance of *`LineagePredictor`*.

The initializer finds the associated workspace and initializes a *`Lineage`* instance. The input_resource is recorded in this lineage. Other methods call the underlying wrapped predictor's methods, with additional functionality as needed (see below).

Parameters

predictor Any sklearn predictor instance. It must have `fit` and `predict` methods.

metrics Either a string naming a metrics type or a subclass of *`Metrics`*. If a string, it should be one of: binary_classification, multiclass_classification, or regression.

input_resource Resource providing the input data to this model. May be specified by name, by a local file path, or via a `ResourceRef`.

resource_resource (optional) Resource where the results are to be stored. May be specified by name, by a local file path, or via a `ResourceRef`. If not specified, will try to infer from the workspace.

model_save_file (optional) Name of file to store a (joblib-formmatted) serialization of the trained model upon completion of the `fit()` method. This should be a relative path, as it is stored under the results resource. If model_save_file is not specified, no model is saved.

workspace_dir (optional) Directory specifying the workspace. Usually can be inferred from the current directory.

verbose If True, print a lot of detailed information about the execution of Data Workspaces.

Example

Here is an example useage of the wrapper, taken from the *Quick Start*:

```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from dataworkspaces.kits.scikit_learn import load_dataset_from_resource
from dataworkspaces.kits.scikit_learn import LineagePredictor

dataset = load_dataset_from_resource('sklearn-digits-dataset')
X_train, X_test, y_train, y_test = train_test_split(
    dataset.data, dataset.target, test_size=0.5, shuffle=False)
classifier = LineagePredictor(SVC(gamma=0.001),
                              metrics='multiclass_classification',
                              input_resource=dataset.resource,
                              model_save_file='digits.joblib')

classifier.fit(X_train, y_train)
score = classifier.score(X_test, y_test)
```

Methods

`fit`(*X*, *y*, **args*, ***kwargs*)

The underlying fit() method of a predictor trains the predictio based on the input data (X) and labels (y).

If the input resource is an api resource, the wrapper captures the hash of the inputs. If `model_save_file` was specified, it also saves the trained model.

`predict`(*X*)

The underlying *`predict()`* method is called directly, without affecting the lineage.

`score`(*X*, *y*, *sample_weight=None*)

This method make predictions from a trained model and scores them according to the metrics specified when instantiated the wrapper.

If the input resource is an api resource, the wrapper captures its hash. The wapper runs the wrapped predictor's *`predict()`* method to generate predictions. A *metrics* object is instantiated to compute the metrics for the predictions and a `results.json` file is written to the results resource. The lineage data is saved and finally the score is computed from the predictions and returned to the caller.

`class dataworkspaces.kits.scikit_learn.Metrics`(*expected*, *predicted*, *sample_weight=None*)

Metrics and its subclasses are convenience classes for sklearn metrics. The subclasses of Matrics are used by `train_and_predict_with_cv()` in printing a metrics report and generating the metrics json file.

`abstract print_metrics`(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*) → None

Print the metrics to a file

`abstract score`() → float

Given the expected and predicted values, compute the metric for this type of predictor, as needed for the predictor's score() method. This is used in the wrapped classes to avoid multiple calls to predict().

`class dataworkspaces.kits.scikit_learn.MulticlassClassificationMetrics`(*expected*, *predicted*, *sample_weight=None*)

Given an array of expected (target) values and the actual predicted values from a classifier, compute metrics that make sense for a multi-class classifier, including accuracy and sklearn's "classification report" showing per-class metrics.

`print_metrics`(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Print the metrics to a file

`score`() → float
Metric for multiclass classification is accuracy

`dataworkspaces.kits.scikit_learn.load_dataset_from_resource`(*resource_name: str*, *subpath: Optional[str] = None*, *workspace_dir: Optional[str] = None*) → sklearn.utils.Bunch

Load a datset (data and targets) from the specified resource, and returns an sklearn-style Bunch (a dictionary-like object). The bunch will include at least three attributes:

- `data` - a NumPy array of shape number_samples * number_features
- `target` - a NumPy array of length number_samples
- `resource` - a `ResourceRef` that provides the resource name and subpath (if any) for the data

Some other attributes that may also be present, depending on the data set:

- `DESCR` - text containing a full description of the data set (for humans)
- `feature_names` - an array of length number_features containing the name of each feature.
- `target_names` - an array containing the name of each target class

Data sets may define their own attributes as well (see below).

Parameters

resource_name The name of the resource containing the dataset.

subpath Optional subpath within the resource where this specific dataset is located. If not specified, the root of the resource is used.

workspace_dir The root directory of your workspace in the local file system. Usually, this can be left unspecified and inferred by DWS, which will search up from the current working directory.

Creating a Dataset

To create a dataset in your resource that is suitable for importing by this function, you simply need to create a file for each attribute you want in the bunch and place all these files in the same directory within your resource. The names of the files should be `ATTRIBUTE.extn` where `ATTRIBUTE` is the attribute name (e.g. `data` or `DESCR`) and `.extn` is a file extension indicating the format. Supported file extensions are:

- `.txt` or `.rst` - text files
- `.csv` - csv files. These are read in using `numpy.loadtxt()`. If this fails because the csv does not contain all numeric data, pandas is used to read in the file. It is then converted back to a numpy array.
- `.csv.gz` or `.csv.bz2` - these are compressed csv files which are treated the same was as csv files (numpy and pandas will automatically uncompress before parsing).
- `.npy` - this a a file containing a serialized NumPy array saved via `numpy.save()`. It is loaded using `numpy.load()`.

1.5.3 TensorFlow

Integration with Tensorflow 1.x and 2.0

This is an experimental API and subject to change.

Wrapping a Karas Model

Below is an example of wrapping one of the standard tf.keras model classes, based on https://www.tensorflow.org/tutorials/keras/basic_classification. Assume we have a workspace already set up, with two resources: a *Source Data* resource of type *api-resource*, which is used to capture the hash of input data as it is passed to the model, and a *Results* resource to keep the metrics. The only change we need to do to capture the lineage from the model is to wrap the model's class, using `add_lineage_to_keras_model_class()`.

Here is the code:

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

from dataworkspaces.kits.tensorflow1 import add_lineage_to_keras_model_class

# Wrap our model class. This is the only DWS-specific change needed.
# We add an optional checkpoint configuration, which will cause checkpoints
# to be written to the workspace's scratch directory and then the best
# checkpoint copied to the results resource.
keras.Sequential = add_lineage_to_keras_model_class(keras.Sequential,
                       checkpoint_config=CheckpointConfig(model='fashion',
                                                          monitor='loss'))

fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)
```

This will create a `results.json` file in the results resource. It will look like this:

```
{
  "step": "test",
  "start_time": "2019-09-26T11:33:22.100584",
  "execution_time_seconds": 26.991521,
```

(continues on next page)

(continued from previous page)

```
  "parameters": {
    "optimizer": "adam",
    "loss_function": "sparse_categorical_crossentropy",
    "epochs": 5,
    "fit_batch_size": null,
    "evaluate_batch_size": null
  },
  "run_description": null,
  "metrics": {
    "loss": 0.3657455060243607,
    "acc": 0.8727999925613403
  }
}
```

Subclassing from a Keras Model

If you subclass from a Keras Model class, you can just use `add_lineage_to-keras_model_class()` as a decorator. Here is an example:

```python
@add_lineage_to_keras_model_class
class MyModel(keras.Model):
  def __init__(self):
    # The Tensorflow documentation tends to specify the class name
    # when calling the superclass __init__ function. Don't do this --
    # it breaks if you use class decorators!
    #super(MyModel, self).__init__()
    super().__init__()
    self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
    self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)

  def call(self, inputs):
    x1 = self.dense1(inputs)
    return self.dense2(x1)

model = MyModel()

import numpy as np

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(np.zeros(20).reshape((5,4)), np.ones(5), epochs=5)
test_loss, test_acc = model.evaluate(np.zeros(16).reshape(4,4), np.ones(4))

print('Test accuracy:', test_acc)
```

Supported datatypes for API Resources

If you are using the *API Resource Type* for your input resource, the model wrapper will hash the incoming data parameters and include the hash values in the data lineage. To compute the hashes, Data Workspaces must access the underlying data representation. The following data types are currently supported:

- NumPy `ndarray`
- Pandas `DataFrame` and `Series`

- Tensorflow `Tensor` and `Dataset`, as well as tuples and dictionaries containing these types. These types supported if you are either running Tensorflow 2.x (graph or eager mode) or 1.x only in eager mode. This restriction is due to the inability to access the underlying tensor representation when Tensorflow is running in graph mode in version 1.x

If you are using another data representation, or running Tensorflow 1.x in graph mode, you can always use a resource type that stores the data in files (e.g. git or local-files) and pass in the input resource name to the wrapper function.

API

class `dataworkspaces.kits.tensorflow.`**`CheckpointConfig`**(*model_name: str*, *monitor: str = 'val_loss'*, *save_best_only: bool = False*, *mode: str = 'auto'*, *save_freq: Union[str, int] = 'epoch'*)

Configuration for checkpoints, to be passed as a parameter to *`add_lineage_to_keras_model_class()`*, instead of directly instantiating `DwsModelChecpoint`.

The checkpoints are initially written under the workspace's scratch space. At the end of training, the best checkpoint is copied to the results resource.

The configuration fields are:

- `model_name` - name of the model to use in checkpoint files
- `monitor` - metric to monitor - defaults to val_loss
- `save_best_only` - if True, only checkpoints better than the previous are kept.
- `mode` - how to determine whether a metric is the "best" - auto, min, or max
- `save_freq` - 'epoch' or an interger

property mode

Alias for field number 3

property model_name

Alias for field number 0

property monitor

Alias for field number 1

property save_best_only

Alias for field number 2

property save_freq

Alias for field number 4

class `dataworkspaces.kits.tensorflow.`**`DwsModelCheckpoint`**(**args: Any*, ***kwargs: Any*)

Subclass of tf.keras.callbacks.ModelCheckpoint which will save checkpoints to the workspace's stratch space and then move the most recent/best checkpoint to the results directory at the end of the run.

You can instantiate this class directly and pass it to the `callbacks` parameter of the model's `fit()` method:

```
model.fit(train_images, train_labels, epochs=10,
          callbacks=[DwsModelCheckpoint('fashion', monitor='loss', save_best_
↪only=True)])
```

You can also pass *`CheckpointConfig`* instance to the *`add_lineage_to_keras_model_class()`* wrapper function.

`dataworkspaces.kits.tensorflow.`**`add_lineage_to_keras_model_class`**(*Cls: type*, *input_resource: Optional[Union[str,* dataworkspaces.utils.lineage_utils.ResourceRef*]] = None*, *results_resource: Optional[Union[str,* dataworkspaces.utils.lineage_utils.ResourceRef*]] = None*, *workspace_dir: Optional[str] = None*, *checkpoint_config: Optional[*dataworkspaces.kits.tensorflow.CheckpointC
= None, *verbose: bool = False*)
→ type

This function wraps a Keras model class with a subclass that overwrites key methods to make calls to the data lineage API.

Parameters:

- `Cls` – the class being wrapped
- `input_resources` – optional list of input resources to this model. Each resource may be specified by name, by a local file path, or via a `ResourceRef`. If no inputs are specified, will try to infer from the workspace.
- `results_resource` – optional resource where the results are to be stored. May be specified by name, by a local file path, or via a `ResourceRef`. if not specified, will try to infer from the workspace.
- `workspace-dir` – Optional directory specifying the workspace. Usually can be inferred from the current directory.
- `checkpoint_config` – Optional instance of *`CheckpointConfig`*, which is used to enable checkpointing on fit and fit_generator()
- `verbose` – If True, print extra debugging information.

The following methods are wrapped:

- `__init__()` - loads the workspace and adds dws-specific class members
- `compile()` - captures the `optimizer` and `loss_function` parameter values
- `fit()` - captures the `epochs` and `batch_size` parameter values; if input is an API resource, capture hash values of training data, otherwise capture input resource name. If the input is an API resource, and it is either a Keras Sequence or a generator, writes the generator and captures the hashes of returned values as it is iterated through.
- `evaluate()` - captures the `batch_size` parameter value; if input is an API resource, capture hash values of test data, otherwise capture input resource name; capture metrics and write them to results resource. If the input is an API resource, and it is either a Keras Sequence or a generator, writes the generator and captures the hashes of returned values as it is iterated through.

1.6 6. Resource Reference

This section provide a little detail on how to use specific resource types. For specific command line options, please see the *Command Reference*.

1.6.1 Git resources

The `git` resource type provides project tracking and management for Git repositories. There are actually two types of git resources supported:

1. A standalone repository. This can be either one controller by the user or, for source data, a 3rd party repository to be treated as a read-only resource.
2. A subdirectory of the data workspace's git repository can be treated as a separate resource. This is especially convenient for small projects, where multiple types of data (source data, code, results, etc.) can be kept in a single repository but versioned independently.

When running the `dws add git ...` commend, the type of repository (standalone vs. subdirectory of the main workspace) is automatically detected. In either case, it is expected that there is a local copy of the repository available when adding it as a resource to the workspace. It is recommended, but not required, to have a remote `origin`, so that the `push`, `pull` and `clone` commands can work with the resource.

Examples

When initializing a new workspace, one can add sub-directory resources for any and each of the resource roles (source-data, code, intermediate-data, and results). This is done via the `--create-resources` option as follows:

```
$ mkdir example-ws
$ cd example-ws/
$ dws init --create-resources=code,results
$ dws init --create-resources=code,results
  Have now successfully initialized a workspace at /Users/dws/code/t/example-ws
  Will now create sub-directory resources for code, results
  Added code to git repository
  Have now successfully Added Git repository subdirectory code in role 'code' to␣
↪workspace
  Added results to git repository
  Have now successfully Added Git repository subdirectory results in role 'results' to␣
↪workspace
  Finished initializing resources:
    code: ./code
    results: ./results
$ ls
code  results
```

Here is an example from the *Quick Start* where we add an entire third party repository to our workspace as a read-only resource. We first clone it into a subdirectory of the workspace and then tell `dws` about it:

```
git clone https://github.com/jfischer/sklearn-digits-dataset.git
dws add git --role=source-data --read-only ./sklearn-digits-dataset
```

1.6.2 Support for Large Files: Git-lfs and Git-fat integration

It can be nice to manage your golden source data in a Git repository. Unfortunately, due to its architecture and focus as a source code tracking system, Git can have significant performance issues with large files. Furthermore, hosting services like GitHub place limits on the size of individual files and on commit sizes. To get around this, various extensions to Git have sprung up. Data Workspaces currently integrates with two of them, git-lfs and git-fat.

Git-lfs

Git-lfs (large file storage) is a utility which interacts with a git hosting service using a special protocol. This protocol is supported by most popular Git hosting services/servers, including GitHub and GitLab. You need to manually install the `git-lfs` executable (see https://git-lfs.github.com for details).

Data Workspaces automatically determines whether a particular git repository is using `git-lfs` by looking for any references to `git-lfs` in a `.gitattributes` within the repository. This is done for both the workspace's metadata repository and any git resources. DWS also will ensure that the user is correctly configured for `git-lfs`, by running `git-lfs install` if the user does not have an associated entry for in their `.gitconfig` file.

We support the following integration points with `git-lfs`:

1. The git repo for the workspace itself can be git-lfs enabled when it is created. This is done through the `--git-lfs-attributes` command line option on `dws init`. See the *Command Reference* entry for details (or the example below).
2. Any `dws push` or `dws pull` of a git-lfs-enabled workspace will automatically call the associated git-lfs command for the workspace's main repo.
3. If you add a git repository as a resource to the workspace, and it has references to `git-lfs` in a `.gitattributes` file, then any `dws push` or `dws pull` commands will automatically call the associated `git-lfs` commands.

Git-fat

Git-fat allows you to store your large files on a host you control that is accessible via `ssh` (or other protocols supported through `rsync`). The large files themselves are hashed and stored on the (remote) server. The metadata for these files is stored in the git repository and versioned with the rest of your git files.

Git-fat is just a Python script, which we ship as a part of the `dataworkspaces` Python package.[1] Running `pip install dataworkspaces` will put `git-fat` into your path and make it available to your `git` commands and `dws` commands.

We support the following integration points with `git-fat`:

1. The git repo for the workspace itself can be git-fat enabled when it is created. This is done through command line options on `dws init`. See the *Command Reference* entry for details (or the example below).
2. Any `dws push` or `dws pull` of a git-fat-enabled workspace will automatically call the associated git-fat command for the workspace's main repo.
3. If you add a git repository as a resource to the workspace, and it has a `.gitfat` file, then any `dws push` or `dws pull` commands will automatically call the associated git-fat commands.
4. As mentioned above, git-fat is included in the dataworkspaces package and installed in your path.

[1] Unfortunately, `git-fat` is written in Python 2, so you will need to have Python 2 available on your system to use it.

Example

Here is an example using git-fat to store all gzipped files of the workspace's main git repo on a remote server.

First, we set up a directory on our remote server to store the large files:

```
fat@remote-server $ mkdir ~/fat-store
```

Now, back on our personal machine, we initialize a workspace, specifying the remote server and that .gz files should be managed by git-fat:

```
local $ mkdir git-fat-example
local $ cd git-fat-example/
local $ dws init --create-resources=source-data \
                 --git-fat-remote=remote-server:/home/fat/fat-store \
                 --git-fat-user=fat --git-fat-attributes='*.gz'
local $ ls
source-data
```

A bit later, we've added some .gz files to our source data resource. We take a snapshot and then `dws push` to the origin:

```
local $ ls source-data
README.txt                   census-state-populations.csv.gz zipcode.csv.gz
local $ dws snapshot s1
local $ dws push # this will also push to the remote fat store
```

If we now go to the remote store, we can see the hashed files:

```
fat@remote-server $ ls fat-store
26f2cac452f70ad91da3ccd05fc40ba9f03b9f48  d9cc0c11069d76fe9435b9c4ca64a335098de2d7
```

Our local workspace has our full files, which can be used by our scripts as-is. However, if you look at the origin repository, you will find the content of each .gz file replaced by a single line referencing the hash. If you clone this repo, you will get the full files, through the magic of git-fat.

1.6.3 Adding resources using rclone

The rclone resource type leverages the rclone command line utility to provide synchronization with a variety of remote data services.

`dws add rclone [options] source-repo target-repo`

dws add rclone adds a remote repository set up using rclone.

We use rclone to set up remote repositories.

Example

We use rclone config to set up a repository pointing to a local directory:

```
$ rclone config show
; empty config

$ rclone config create localfs local unc true
```

The configuration file (typically at `~/.config/rclone/rclone.conf`) now looks like this:

```
[localfs]
type = local
config_automatic = yes
unc = true
```

Next, we use the backend to add a repository to dws:

```
$ dws add rclone --role=source-data my_local_files:/Users/rupak/tmp tmpfiles
```

This creates a local directory tmpfiles and copies the contents of /Users/rupak/tmp to it.

Similarly, we can make a remote S3 bucket:

```
$ rclone config
mbk-55-51:docs rupak$ rclone --config=rclone.conf config
Current remotes:

Name                 Type
====                 ====
localfs              local

e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
q) Quit config
e/n/d/r/c/s/q> n
name> s3bucket
Type of storage to configure.

# Pick choice 4 for S3 and configure the bucket
...
# set configuration parameters
```

Once the S3 bucket is configured, we can get files from it:

```
$ dws add rclone --role=source-data s3bucket:mybucket s3files
```

Configuration Files

By default, we use the default configuration file used by rclone. This is the file printed out by:

```
$ rclone config file
```

and usually resides in `$HOME/.config/rclone/rclone.conf`

However, you can specify a different configuration file:

```
$ dws add rclone --config=/path/to/configfile --role=source-data localfs:/Users/rupak/
↪tmp tmpfiles
```

In this case, make sure the config file you are using has the remote `localfs` defined.

rclone synchronization options

Data Workspaces uses `rclone` for uni-directional synchronization. Either the workspace or the `rclone` remote is the *master* and the other side is the *replica*. All changes must be made on the master side and the replica side treats the data as read-only. The roles of the workspace and the remote are determined with the `dws add rclone` command via the `-master` option. It must hve one of three values:

- `none` (the default) means that no action is taken for pushes and pulls. The user is responsible for manually calling rclone for synchronization. Use this option if you want full control over your synchronization or need to change the direction of the synchronization based on the situation.
- `remote` means that pulls will be done, but not pushes. The data is treated as read-only from the perspective of the workspace.
- `local` means that the local system is the master and the remote the replica. Pushes will be done for this resource, but not pulls.

When first adding the resource or cloning to a new machine, if the local directory does not exist, and `remote` or `none` were specified, the contents of the remote will copied down to the local directory.

The `--sync-mode` option to `dws add rclone` will also impact the synchonization behavior. This option determines the `rclone` command used when synchronization data. The following two options are supported:

- `copy` - files are added or overwritten without deleting any files present at the replica. This is the default and the "safer" option.
- `sync` - files at the replica are removed if they are no longer present at the master. If the master both adds and removes files, this option ensures that the master and replica contain the same set of files.

1.6.4 S3 Resources

The S3 resource type provides a resource interface for an Amazon AWS S3 bucket. Unlike the Git, rclone, and local files resource types, S3 resources do not store files locally. Instead, files are accessed via the `ResourceFileSystem` interface defined in `dataworkspaces.api`. The S3 bucket versioning capability is leveraged to support accessing older versions of files associated with a snapshot.

Installation

The S3 dependencies are not included with DWS by default. To install with the required dependencies, specify the "s3" extra as follows:

```
pip install dataworkspaces[s3]
```

Adding an S3 Resource

To add an S3 resource to your workspace, first ensure that versioning is enabled for your bucket on AWS. Next, configure the AWS credentials on your local machine using Amazon's `aws` command line tool. Then, run the `dws add` command as follows, where `BUCKET_NAME` is the name of your bucket, and ROLE is one of source-data, code, or intermediate-data:

```
dws add s3 --role ROLE BUCKET_NAME
```

Accessing Files

After adding an S3 resource, you can get a `ResourceFileSystem` instance via a call to `get_filesystem_for_resource()`. Here is an example:

```
import csv
from dataworkspaces.api import get_filesystem_for_resource

fs = get_filesystem_for_resource('my-s3-resource')

# get a list of the top level files and directories
files = fs.ls('')

# open a csv file for reading
with fs.open('myfile.csv' 'r') as f:
    reader = csv.reader(f)
```

Snapshots

When you take a snapshot of an S3 resource, a file is created in the bucket with the key `.snapshots/HASH.json.gz`, where HASH is the hash code of the snapshot. This file contains a mapping between each file currently active in the bucket and its version id. The file is also cached locally in the workspace, but not checked into the workspace's Git repository.

A file containing the hash code associated with the snapshot is also created at `.dataworkspaces/scratch/RESOURCE_NAME/current_snapshot.txt`. When this file is present, the filesystem interface described in the previous section will return the set of files and versions as of the snapshot rather than the latest in the bucket. When a prior snapshot is restored, the hash associated with that snapshot is written to the `current_snapshot.txt` file, causing that snapshot to be used for determining files and file versions.

When a snapshot is active for the S3 resource, the resource becomes read-only. If you manually remove the `current_snapshot.txt` file, the latest versions of each file become visible through the API.

Feedback Requested

This is the first major release with the S3 resource functionality. We would appreciate your feedback on how you want to use S3 resources in your data science projeccts. Thanks!

1.7 7. Internals: Developer's Guide

This section is a guide for people working on the development of Data Workspaces or people who which to extend it (e.g. through their own resource types or kits).

1.7.1 Installation and setup for development

In summary:

1. Install Python 3 via the Anaconda distribution.
2. Make sure you have the `git` and `make` utilties on your system.
3. Create a virtual environment called `dws` and activate it.
4. Install `mypy` via `pip`.
5. Clone the main `data-workspaces-python` repo.
6. Install the `dataworkpaces` package into your virtual environment.
7. Download and configure `rclone`.
8. Run the tests.

Here are the details:

We recommend using Anaconda3 for development, as it can be easily installed on Mac and Linux and includes most of the packages you will need for development and data science projects. You will also need some basic system utilities: `git` and `make` (which may already be installed).

Once you have Anaconda3 installed, create a virtual environment for your work:

```
conda create --name dws
```

To activate the environment:

```
conda activate dws
```

You will need the mypy type checker, which is run as part of the tests. It is best to install via pip to get the latest version (some older versions may be buggy). Once you have activated your environment, `mypy` may be installed as follows:

```
pip install mypy
```

Next, clone the Data Workspaces main source tree:

```
git clone git@github.com:data-workspaces/data-workspaces-core.git
```

Now, we install the data workspaces library, via `pip`, using an editable mode so that our source tree changes are immediately visible:

```
cd data-workspaces-core
pip install --editable `pwd`
```

With this setup, you should not have to configure PYTHONPATH.

Next, we install `rclone`, a file copying utility used by the *rclone resource*. You can download the latest `rclone` executable from http://rclone.org. Just make sure that the executable is available in your executable path. Alternatively, on Linux, you can install `rclone` via your system's package manager. To configure `rclone`, see the instructions *here* in the Resource Reference.

Now, you should be ready to run the tests:

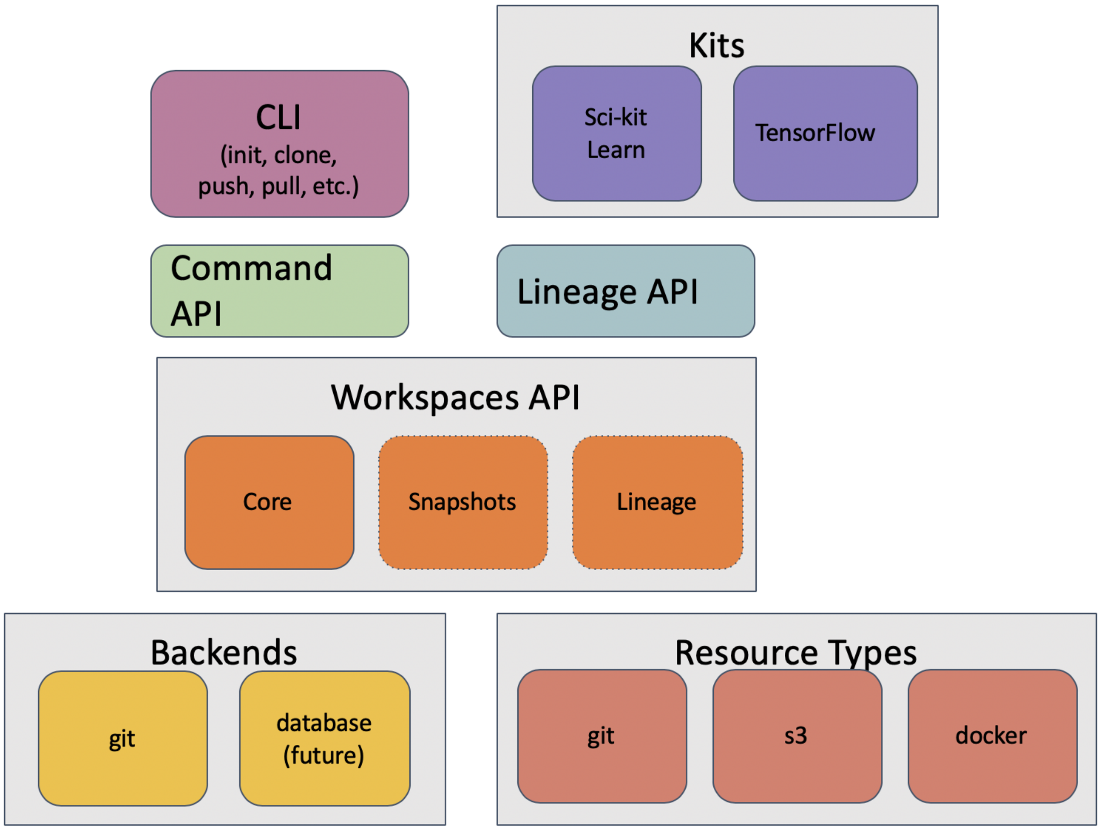
```
cd tests
make test
```

The tests will print a lot to the console. If all goes well, it should end with something like this:

```
----------------------------------------------------------------------
Ran 40 tests in 23.664s

OK
```

1.7.2 Overall Design

Here is a block diagram of the system architecture:



The core of the system is the Workspaces API, which is located in `dataworkspaces/workspace.py`. This API provides a base `Workspace` classes for the workspace and a base `Resource` class for resources. There are also several mixin classes which define extensions to the basic functionality. See *Core Workspace and Resource API* for details.

The `Workspace` class has one or more *backends* which implement the storage and management of the workspace metadata. Currently, there is only one complete backend, the *git* backend, which stores its metadata in a git repository.

Independent of the workspace backend are *resource types*, which provide concrete implementations of the the `Resource` base class. These currently include resource types for git, git subdirectories, rclone, and local files.

Above the workspace API sits the *Command API*, which implements command functions for operations like init, clone, push, pull, snapshot and restore. There is a thin layer above this for programmatic access (`dataworkspaces.api`) as well as a full command line interface, implmented using the click package (see `dataworkspaces.dws`).

The *Lineage API* is also implemented on top of the basic workspace interface and provides a way for pipeline steps to record their inputs, outputs, and code dependencies.

Finally, *kits* provide integration with user-facing libraries and applications, such as Scikit-learn and Jupyter notebooks.

Code Layout

The code is organized as follows:

- `dataworkspaces/`
  - `api.py` - API to run a subset of the workspace commands from Python. This is useful for building integrations.
  - `dws.py` - the command line interface
  - `errors.py` - common exception class definitions
  - `lineage.py` - the generic lineage api
  - `workspace.py` - the core api for workspaces and resources
  - `backends/` - implementations of workspace backends
  - `utils/` - lower level utilities used by the upper layers
  - `resources/` - implementations of the resource types
  - `commands/` - implementations of the individual dws commands
  - `third_party/` - third-party code (e.g. git-fat)
  - `kits/` - adapters to specific external technologies

Git Database Layout

When using the *git backend*, a data workspace is contained within a Git repository. The metadata about resources, snapshots and lineage is stored in the subdirectory `.dataworkspace`. The various resources can be other subdirectories of the workspace's repository or may be external to the main Git repo.

The layout for the files under the `.dataworkspace` directory is as follows:

- `.dataworkspace/`
  - `config.json` - overall configuration (e.g. workspace name, global params)
  - `local_params.json` - local parameters (e.g. hostname); not checked into git
  - `resources.json` - lists all resources and their config parameters
  - `resource_local_params.json` - configuration for resources that is local to this machine (e.g. path to the resource); not checked into git
  - `current_lineage/` - contains lineage files reflecting current state of each resource; not checked into git

  - `file/` - contains metadata for *local files* based resources; in particular, has the file-level hash information for snapshots
  - `snapshots/` - snapshot metadata
    * `snapshot-<HASHCODE>.json` - lists the hashes for each resource in the snapshot. The hash of this file is the hash of the overall snapshot.
    * `snapshot_history.json` - metadata for the past snapshots
  - `snapshot_lineage/` - contains lineage data for past snapshots
    * `<HASHCODE>/` - directory containing the current lineage files at the time of the snapshot associated with the hashcode. Unlike `current_lineeage`, this is checked into git.

In designing the workspace database, we try to follow the following guidelines:

1. Use JSON as our file format where possible - it is human readable and editable and easy to work with from within Python.
2. Local or uncommitted state is not stored in Git (we use .gitignore to keep the files outside of the repo). Such files are initialized by `dws init` and `dws clone`.
3. Avoid git merge conflicts by storing data in seperate files where possible. For example, the resources.json file should really be broken up into one file per resource, stored under a common directory (see issue #13).
4. Use git's design as an inspiration. It provides an efficient and flexible representation.

1.7.3 Command Design

The bulk of the work for each command is done by the core Workspace API and its backends. The command fuction itself (`dataworkspaces.commands.COMMAND_NAME`) performs parameter-checking, calls the associated parts of Workspace API, and handles user interactions when needed.

1.7.4 Resource Design

Resources are orthoginal to commands and represent the collections of files to be versioned.

A resource may have one of four roles:

1. **Source Data Set** - this should be treated read-only by the ML pipeline. Source data sets can be versioned.
2. **Intermediate Data** - derived data created from the source data set(s) via one or more data pipeline stages.
3. **Results** - the outputs of the machine learning / data science process.
4. **Code** - code used to create the intermediate data and results, typically in a git repository or Docker container.

The treatment of resources may vary based on the role. We now look at resource functionality per role.

Source Data Sets

We want the ability to name source data sets and swap them in and out without changing other parts of the workspace. This still needs to be implemented.

Intermediate Data

For intermediate data, we may want to delete it from the current state of the workspace if it becomes out of date (e.g. a data source version is changed or swapped out). This still needs to be implemented.

Results

In general, results should be additive.

For the `snapshot` command, we move the results to a specific subdirectory per snapshot. The name of this subdirectory is determined by a template that can be changed by setting the parameter `results.subdir`. By default, the template is: `{DAY}/{DATE_TIME}-{USER}-{TAG}`. The moving of files is accomplished via the method `results_move_current_files(rel_path, exclude)` on the *Resource <resources>* class. The `snapshot()` method of the resource is still called as usual, after the result files have been moved.

Individual files may be excluded from being moved to a subdirectory. This is done through a configuration command. Need to think about where this would be stored – in the resources.json file? The files would be passed in the exclude set to `results_move_current_files`.

If we run `restore` to revert the workspace to an older state, we should not revert the results database. It should always be kept at the latest version. This is done by always putting results resources into the leave set, as if specified in the `--leave` option. If the user puts a results resource in the `--only` set, we will error out for now.

1.7.5 Integration API

The module `dataworkspaces.api` provides a simplified, high level programmatic inferface to Data Workspaces. It is for integration with third-party tooling.

This is an API for selected Data Workspaces management functions.

class `dataworkspaces.api.`**`ResourceFileSystem`**(*resource: dataworkspaces.workspace.FileResourceMixin*)
subset of fsspec supported by our file resources.

class `dataworkspaces.api.`**`ResourceInfo`**(*name: str*, *role: str*, *resource_type: str*, *local_path: Optional[str]*)
Named tuple representing the results from a call to *`get_resource_info()`*.

property local_path
Alias for field number 3

property name
Alias for field number 0

property resource_type
Alias for field number 2

property role
Alias for field number 1

class `dataworkspaces.api.`**`SnapshotInfo`**(*snapshot_number: int*, *hashval: str*, *tags: List[str]*, *timestamp: str*, *message: str*, *metrics: Optional[Dict[str, Any]]*)
Named tuple represneting the results from a call to *`get_snapshot_history()`*

property hashval
Alias for field number 1

property message
Alias for field number 4

property metrics
Alias for field number 5

property snapshot_number
Alias for field number 0

property tags
Alias for field number 2

property timestamp
Alias for field number 3

`dataworkspaces.api.`**get_api_version**()
The API version is maintained independently of the overall DWS version. It should be more stable.

`dataworkspaces.api.`**get_filesystem_for_resource**(*name: str*, *workspace_uri_or_path: Optional[str] = None*, *verbose: bool = False*) → Optional[*dataworkspaces.api.ResourceFileSystem*]
Get the a filesystem-like object for the named resource. If it isn't a FileResource, returns None.

`dataworkspaces.api.`**get_local_path_for_resource**(*name: str*, *workspace_uri_or_path: Optional[str] = None*, *verbose: bool = False*) → Optional[str]
If a local path is available for this resource, return it. Otherwise, return None.

`dataworkspaces.api.`**get_resource_info**(*workspace_uri_or_path: Optional[str] = None*, *verbose: bool = False*) → List[*dataworkspaces.api.ResourceInfo*]
Returns a list of ResourceInfo instances, describing the resources defined for this workspace.

`dataworkspaces.api.`**get_results**(*workspace_uri_or_path: Optional[str] = None*, *tag_or_hash: Optional[str] = None*, *resource_name: Optional[str] = None*, *verbose: bool = False*) → Optional[Tuple[Dict[str, Any], str]]
Get a results file as a parsed json dict. If no resource or snapshot is specified, searches all the results resources for a file. If a snapshot is specified, we look in the subdirectory where the resuls have been moved. If no snapshot is specified, and we don't find a file, we look in the most recent snapshot.

Returns a tuple with the results and the logical path (resource:/subpath) to the results. If nothing is found, returns None.

`dataworkspaces.api.`**get_snapshot_history**(*workspace_uri_or_path: Optional[str] = None*, *reverse: bool = False*, *max_count: Optional[int] = None*, *verbose: bool = False*) → Iterable[*dataworkspaces.api.SnapshotInfo*]
Get the history of snapshots, starting with the oldest first (unless :reverse: is True). Returns a list of SnapshotInfo instances, containing the snapshot number, hash, tag, timestamp, and message. If :max_count: is specified, returns at most that many snapshots.

`dataworkspaces.api.`**get_version**()
Get the version string for the installed version of Data Workspaces

`dataworkspaces.api.`**make_lineage_graph**(*output_file: str*, *workspace_uri_or_path: Optional[str] = None*, *resource_name: Optional[str] = None*, *tag_or_hash: Optional[str] = None*, *width: int = 1024*, *height: int = 800*, *verbose: bool = False*) → None
Write a lineage graph as an html/javascript page to the specified file.

dataworkspaces.api.**make_lineage_table**(*workspace_uri_or_path: Optional[str] = None*, *tag_or_hash: Optional[str] = None*, *verbose: bool = False*) → Iterable[Tuple[str, str, str, Optional[List[str]]]]

Make a table of the lineage for each resource. The columns are: ref, lineage type, details, inputs

dataworkspaces.api.**restore**(*tag_or_hash: str*, *workspace_uri_or_path: Optional[str] = None*, *only: Optional[List[str]] = None*, *leave: Optional[List[str]] = None*, *verbose: bool = False*) → int

Restore to a previous snapshot, identified by either its hash or its tag (if one was specified). Parameters:

- `only` - an optional list of resources to store. If specified all other resources will be left as-is.
- `leave` - an optional list of resource to leave as-is. Both `only` and `leave` should not be specified together.

Returns the number of resources changed.

dataworkspaces.api.**take_snapshot**(*workspace_uri_or_path: Optional[str] = None*, *tag: Optional[str] = None*, *message: str = ''*, *verbose: bool = False*) → str

Take a snapshot of the workspace, using the tag and message, if provided. Returns the snapshot hash (which can be used to restore to this point).

1.7.6 Core Workspace API

Here is the detailed documentation for the Core Workspace API, found in `dataworkspaces.workspace`.

Main definitions of the workspace abstractions

Workspace backends, like git, subclass from the `Workspace` base class. Resource implementations (e.g. local files or git resource) subclass from the `Resource` base class.

Optional capabilities for both workspace backends and resource backends are defined via abstract *mixin* classes. These classes do not inherit from the base workspace/resource classes, to avoid issues with multiple inheritance.

Complex operations involving resources use the following pattern, where COMMAND is the command and CAPABILITY is the capability needed to perform the command:

```
class CAPABILITYWorkspaceMixin:
  ...
  def _COMMAND_precheck(self, resource_list:List[CAPABILITYResourceMixin]) -> None:
    # Backend can override to add more checks
    for r in resource_list:
      r.COMMAND_precheck()

  def COMMAND(sef resource_list:List[CAPAIBILITYResourceMixin]) -> None:
    self._COMMAND_precheck(resource_list)
    ...
    for r in resource_list:
      r.COMMAND()
```

The module `dataworkspaces.commands.COMMAND` should look like this:

```
def COMMAND_command(workspace, ...):
  if not isinstance(workspace, CAPABILITYWorkspaceMixin):
    raise ConfigurationError("Workspace %s does not support CAPABILITY"%
                             workspace.name)
  mixin = cast(CAPABILITYWorkspaceMixin, workspace)

```

(continues on next page)

(continued from previous page)

```
... error checking ...

resource_list = ...

workspace.COMMAND(resource_list)
workspace.save("Completed command COMMAND")
```

Core Classes

class dataworkspaces.workspace.**Workspace**(*name: str*, *dws_version: str*, *batch: bool = False*, *verbose: bool = False*)

add_resource(*name: str*, *resource_type: str*, *role: str*, **args*, ***kwargs*) → *dataworkspaces.workspace.Resource*

Add a resource to the repository for tracking.

abstract as_lineage_ws() → *dataworkspaces.workspace.SnapshotWorkspaceMixin*

If this workspace supports snapshots and lineage, cast it to a SnapshotWorkspaceMixin. Otherwise, raise an NotSupportedError exception.

abstract as_snapshot_ws() → *dataworkspaces.workspace.SnapshotWorkspaceMixin*

If this workspace supports snapshots, cast it to a SnapshotWorkspaceMixin. Otherwise, raise an NotSupportedError exception.

batch

attribute: True if input from user should be avoided (bool)

clone_resource(*name: str*) → *dataworkspaces.workspace.LocalStateResourceMixin*

Instantiate the resource locally. This is used in cases where the resource has local state.

dws_version

attribute: Version of dataworkspaces that was used to create the workspace (str)

get_global_param(*param_name: str*) → Any

Returns the value of the global param if set, otherwise the default. If the param is not set, returns the default value. If the param is not defined throws ParamNotFoundError.

abstract get_instance() → str

Return a unique identifier for this instance of the workspace. For lineage tracking, it is assumed that only one pipeline is running at a time in an instance. If the workspace exists on a local filesystem, then it should correspond to the machine and path where the workspace resides. Typically, some combination of hostname and user are sufficient.

Uniquenes of the instance is important for things like naming the results snapshot subdirectories.

get_local_param(*param_name: str*) → Any

Returns the value of the local param if set, otherwise the default. If the param is not set, returns the default value. If the param is not defined throws ParamNotFoundError.

get_names_for_resources_that_need_to_be_cloned() → Iterable[str]

Find all the resources that have local state, but no local parameters (not even an empty dict). These needed to be cloned. This is to be called during the pull() command.

get_names_of_resources_with_local_state() → Iterable[str]

Return an iterable of the resource names in the workspace that have local state.

`get_resource`(*name: str*) → *dataworkspaces.workspace.Resource*
Get the associated resource from the workspace metadata.

`abstract get_resource_names`() → Iterable[str]
Return an iterable of resource names. The names should be returned in a consistent order, specifically the order in which they were added to the workspace. This supports backwards compatilbity for operations like snapshots.

`get_resource_role`(*resource_name*) → str
Get the role of a resource without having to instantiate it.

`get_resource_type`(*resource_name*) → str
Get the type of a resource without having to instantiate it.

`get_resources`() → Iterable[*dataworkspaces.workspace.Resource*]
Iterate through all the resources

`abstract get_scratch_directory`() → str
Return an absolute path for the local scratch directory to be used by this workspace.

`abstract get_workspace_local_path_if_any`() → Optional[str]
If the workspace maintains local state and has a "home" directory, return it. Otherwise, return None.

This is useful for things like providing defaults for resource local paths or providing special handling for resources enclosed in the workspace (e.g. GitRepoResource vs. GitSubdirResource)

`map_local_path_to_resource`(*path: str*, *expecting_a_code_resource: bool = False*) → *dataworkspaces.utils.lineage_utils.ResourceRef*
Given a path on the local filesystem, map it to a resource and the path within the resource. Raises PathNotAResourceError if no match is found.

Note: this does not check whether the path already exists.

`name`
attribute: A short name for this workspace (str)

`abstract save`(*message: str*) → None
Save the current state of the workspace

`set_global_param`(*name: str*, *value: Any*) → None
Validate and set a global parameter. Setting does not necessarily take effect until save() is called

`set_local_param`(*name: str*, *value: Any*) → None
Validate and set a local parameter. Setting does not necessarily take effect until save() is called

`suggest_resource_name`(*resource_type: str*, *role: str*, **args*)
Given the arguments passed in for creating a resource, suggest a (unique) name for the resource.

`validate_local_path_for_resource`(*proposed_resource_name: str*, *proposed_local_path: str*) → None
When creating a resource, validate that the proposed local path is usable for the resource. By default, this checks existing resources with local state to see if they have conflicting paths and, if a local path exists for the workspace, whether there is a conflict (the entire workspace cannot be used as a resource path).

Subclasses may want to add more checks. For subclasses that do not support *any* local state, including in resources, they can override the base implementation and throw an exception.

`validate_resource_name`(*resource_name: str*, *subpath: Optional[str] = None*, *expected_role: Optional[str] = None*) → None
Validate that the given resource name and optional subpath are valid in the current state of the workspace. Otherwise throws a ConfigurationError.

`verbose`
attribute: Print detailed logging (bool)

class `dataworkspaces.workspace.`**`ResourceRoles`**

This class defines constants for the four resource roles.

`CODE = 'code'`

`INTERMEDIATE_DATA = 'intermediate-data'`

`RESULTS = 'results'`

`SOURCE_DATA_SET = 'source-data'`

class `dataworkspaces.workspace.`**`Resource`**(*resource_type: str*, *name: str*, *role: str*, *workspace: dataworkspaces.workspace.Workspace*)

Base class for all resources

`get_params`() → Dict[str, Any]

Get the parameters that define the configuration of the resource globally.

`has_results_role`()

`is_exported`() → bool

Returns True if this resource has an export parameter and it is True.

`is_imported`() → bool

Returns True if this resource has an imported parameter and it is True.

`name`

attribute: unique name for this resource within the workspace (str)

`resource_type`

attribute: name for this resource's type (e.g. git, local-files, etc.) (str)

`role`

Role of the resource, one of *`ResourceRoles`*

abstract `validate_subpath_exists`(*subpath: str*) → None

Validate that the subpath is valid within this resource. Otherwise should raise a ConfigurationError.

`workspace`

attribute: The workspace that contains this resource (Workspace)

`dataworkspaces.workspace.`**`RESOURCE_ROLE_CHOICES = ['source-data', 'intermediate-data', 'code', 'results']`**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Factory Classes and Functions

class `dataworkspaces.workspace.`**`WorkspaceFactory`**

This class collects the various ways of instantiating a workspace: creating from an existing one, initing a new one, and cloning into a new environment.

Each backend should implement a subclass and provide a singleton instance as the FACTORY member of the module.

abstract static `clone_workspace`(*local_params: Dict[str, Any]*, *batch: bool*, *verbose: bool*, **args*) → *dataworkspaces.workspace.Workspace*

Clone an existing workspace into the local environment. Note that hostname is used as an instance identifier (TODO: make this more generic).

This only clones the workspace itself, any local state resources should be cloned separately.

If a workspace has no local state, this factory method might not do anything.

abstract static load_workspace(*batch: bool*, *verbose: bool*, *parsed_uri: urllib.parse.ParseResult*) → *dataworkspaces.workspace.Workspace*

Instantiate and return a workspace.

`dataworkspaces.workspace.`**load_workspace**(*uri: str*, *batch: bool*, *verbose: bool*) → *dataworkspaces.workspace.Workspace*

Given a requested workspace backend, and backend-specific parameters, instantiate and return a workspace. The workspace is specified by a uri, where the backend-type is the scheme and rest is interpreted by the backend.

The backend name / scheme is used to load a backend module whose name is dataworkspaces.backends.SCHEME.

`dataworkspaces.workspace.`**find_and_load_workspace**(*batch: bool*, *verbose: bool*, *uri_or_local_path: Optional[str] = None*) → *dataworkspaces.workspace.Workspace*

This tries to find the workspace and load it. There are three cases:

1. If uri_or_local_path is a uri, we call load_workspace() directly
2. If uri_or_local_path is specified, but not a uri, we interpret it as a local path and try to instantitate a git-backend workspace at that location in the loca filesystem.
3. If uri_or_local_path is not specified, we start at the current directory and search up the directory tree until we find something that looks like a git backend workspace.

TODO: In the future, this should also look for a config file that might specify the workspace or list workspaces by name.

`dataworkspaces.workspace.`**init_workspace**(*backend_name: str*, *workspace_name: str*, *hostname: str*, *batch: bool*, *verbose: bool*, *scratch_dir: str*, **args*, ***kwargs*) → *dataworkspaces.workspace.Workspace*

Given a requested workspace backend, and backend-specific parameters, initialize a new workspace, then instantitate and return it.

A backend name is a module name. The module should have an init_workspace() function defined.

TODO: the hostname should be generalized as an "instance name", but we also need backward compatibility. TODO: is this function now redundant? Compare to `load_workspace()`.

class `dataworkspaces.workspace.`**ResourceFactory**

Abstract factory class to be implemented for each resource type.

abstract clone(*params: Dict[str, Any]*, *workspace:* dataworkspaces.workspace.Workspace) → *dataworkspaces.workspace.LocalStateResourceMixin*

Instantiate a local copy of the resource that came from the remote origin. We don't yet have local params, since this resource is not yet on the local machine. If not in batch mode, this method can ask the user for any additional information needed (e.g. a local path). In batch mode, should either come up with a reasonable default or error out if not enough information is available.

abstract from_command_line(*role: str*, *name: str*, *workspace:* dataworkspaces.workspace.Workspace, **args*, ***kwargs*) → *dataworkspaces.workspace.Resource*

Instantiate a resource object from the add command's arguments

abstract from_json(*params: Dict[str, Any]*, *local_params: Dict[str, Any]*, *workspace:* dataworkspaces.workspace.Workspace) → *dataworkspaces.workspace.Resource*

Instantiate a resource object from saved params and local params

abstract has_local_state() → bool

Return true if this resource has local state and needs a clone step the first time it is used.

abstract suggest_name(*workspace:* dataworkspaces.workspace.Workspace, *role: str*, **args*) → str
Given the arguments passed in to create a resource, suggest a name for the case where the user did not provide one via –name. This will be used by suggest_resource_name() to find a short, but unique name for the resource.

Mixins for Files and Local State

class dataworkspaces.workspace.**FileResourceMixin**
This is a mixin to be implemented by resources which provide a hierarchy of files. Examples include a git repo, local filesystem, or S3 bucket. A database would be a resource that does NOT implement this API.

abstract add_results_file(*data: Union[Dict[str, Any], List[Any]]*, *rel_dest_path: str*) → None
Save JSON results data to the specified path in the resource. Note that, although this is usually used for results role resources, it could also be used for intermediate-data resources if they are exported (causing the lineage file to be written to the resource).

TODO: this is used for both results and lineage files. Perhaps we should either rename it to something like add_json_file() or create a separate call for lineage.

abstract delete_file(*rel_path: str*) → None
Delete a file from the resource. If the resource is read-only or otherwise does not support modifications, should throw a NotSupportedError.

abstract does_subpath_exist(*subpath: str*, *must_be_file: bool = False*, *must_be_directory: bool = False*) → bool
Return True the subpath is valid within this resource, False otherwise. If must_be_file is True, return True only if the subpath corresponds to content. If must_be_directory is True, return True only if the subpath corresponds to a directory.

abstract ls(*rel_path: str*) → List[str]
List the files under the relative path (use empty string for root)

abstract open(*rel_path: str*, *mode: str*)
Returns a file like object in the specified mode.

abstract read_results_file(*subpath: str*) → Dict[str, Any]
Read and parse json results data from the specified path in the resource. If the path does not exist or is not a file throw a ConfigurationError.

abstract results_copy_current_files(*rel_dest_root: str*, *exclude_files: Set[str]*, *exclude_dirs_re: Pattern*) → None
A snapshot is being taken, and we want to copy the files in the resource to the relative subdirectory rel_dest_root. We should exclude the files in the set exclude_files and exclude any directories matching exclude_dirs_re (e.g. the directory to which the files are being moved).

By default results_move_current_files() is called, but the copy is used when we export the resource.

abstract results_move_current_files(*rel_dest_root: str*, *exclude_files: Set[str]*, *exclude_dirs_re: Pattern*) → None
A snapshot is being taken, and we want to move the files in the resource to the relative subdirectory rel_dest_root. We should exclude the files in the set exclude_files and exclude any directories matching exclude_dirs_re (e.g. the directory to which the files are being moved).

abstract upload_file(*src_local_path: str*, *rel_dest_path: str*) → None
Copy a local file to the specified path in the resource. This may be a local copy or an upload, depending on the resource implmentation

class dataworkspaces.workspace.**LocalStateResourceMixin**
Mixin for the resource api for resources with local state that need to be "cloned"

get_local_params() → Dict[str, Any]
Get the parameters that define any local configuration of the resource (e.g. local filepaths)

abstract **get_local_path_if_any**() → Optional[str]
If the resource has an associated local path on the system, return it. Othewise, return None. Even if it has local state, this might not be a file-based resource. Thus, the return value is an Optional string.

abstract **pull**()
Update this resource with the latest changes from the remote origin.

abstract **pull_precheck**()
Perform any prechecks before updating this resource from the remote origin.

abstract **push**()
Upload this resource's changes to the remote origin.

abstract **push_precheck**()
Perform any prechecks before uploading this resource's changes to the remote origin.

validate_subpath_exists(*subpath: str*) → None
Validate that the subpath is valid within this resource. Default implementation checks the local filesystem if any. If the resource is not file-based, then the subclass should override this method to implement the check.

Mixins for Synchronized and Centralized Workspaces

Workspace backends should inherit from one of either `SyncedWorkspaceMixin` or `CentralWorkspaceMixin`.

class dataworkspaces.workspace.**SyncedWorkspaceMixin**
This mixin is for workspaces that support synchronizing with a master copy via push/pull operations.

abstract **publish**(**args*) → None
Make a local repo available at a remote location. For example, we may make it available on GitHub, GitLab or some similar service.

pull_resources(*resource_list: List[*dataworkspaces.workspace.LocalStateResourceMixin*]*) → None
Download latest updates from remote origin. By default, includes any resources that support syncing via the LocalStateResourceMixin.

Note that this does not handle the actual workspace pull or the cloning of new resources.

abstract **pull_workspace**() → *dataworkspaces.workspace.SyncedWorkspaceMixin*
Pull the workspace itself and return a new workspace object reflecting the latest state changes.

push(*resource_list: List[*dataworkspaces.workspace.LocalStateResourceMixin*]*) → None
Upload updates to remote origin.

Backend subclass also needs to handle syncing of the workspace itself. If this is called with an empty set of resources, then we are just syncing the workspace. Pushing the workspace should include pushing of any new resources.

class dataworkspaces.workspace.**CentralWorkspaceMixin**
This mixin is for workspaces that have a central store and do not need synchronization of the workspace itself. They still may need to sychronize individual resources.

abstract **get_resources_that_need_to_be_cloned**() → List[str]
Return a list of resources with local state that are not present in the local system. This is used after a pull to clone these resources.

pull_resources(*resource_list: List[*dataworkspaces.workspace.LocalStateResourceMixin*]*) → None
: Download latest resource updates from remote origin for resources that support syncing via the LocalStateResourceMixin.

push_resources(*resource_list: List[*dataworkspaces.workspace.LocalStateResourceMixin*]*) → None
: Upload resource updates to remote origin.

Mixins for Snapshot Functionality

To support snapshots, the interfaces defined by `SnapshotWorkspaceMixin` and `SnapshotResourceMixin` should be implmented by workspace backends and resources, respectively. `SnapshotMetadata` defines the metadata to be stored for each snapshot.

class dataworkspaces.workspace.**SnapshotMetadata**(*hashval: str*, *tags: List[str]*, *message: str*, *hostname: str*, *timestamp: str*, *relative_destination_path: str*, *restore_hashes: Dict[str, Optional[str]]*, *metrics: Optional[Dict[str, Any]] = None*, *updated_timestamp: Optional[str] = None*)
: The metadata we store for each snapshot (in addition to the manifest). relative_destination_path refers to the path used in resources that copy their current state to a subdirectory for each snapshot.

  static from_json(*data: Dict[str, Any]*) → *dataworkspaces.workspace.SnapshotMetadata*

  has_tag(*tag*)

  matches_partial_hash(*partial_hash*)
  : A partial hash matches if the full hash starts with it, normalizing to lower case.

  to_json() → Dict[str, Any]

class dataworkspaces.workspace.**SnapshotWorkspaceMixin**
: Mixin class for workspaces that support snapshots and restores.

  delete_snapshot(*hash_val: str*, *include_resources=False*) → None
  : Given a snapshot hash, delete the entry from the workspace's metadata. If include_resources is True, then delete any data from the associated resources (e.g. snapshot subdirectories).

  abstract get_lineage_store() → dataworkspaces.utils.lineage_utils.LineageStore
  : Return the store for lineage data. If this workspace backend does not support lineage for some reason, the call should raise a ConfigurationError.

  get_most_recent_snapshot() → Optional[*dataworkspaces.workspace.SnapshotMetadata*]
  : Helper function to return the metadata for the most recent snapshot (by timestamp). Returns None if no snapshot found

  abstract get_next_snapshot_number() → int
  : Return a number that can be used for this snapshot. For a given local copy of thw workspace, it is guaranteed to be unique and increasing. It is not guarenteed to be globally unique (need to combine with hostname to get that).

  abstract get_snapshot_by_partial_hash(*partial_hash: str*) → *dataworkspaces.workspace.SnapshotMetadata*
  : Given a partial hash for the snapshot, find the snapshot whose hash starts with this prefix and return the metadata asssociated with the snapshot.

  abstract get_snapshot_by_tag(*tag: str*) → *dataworkspaces.workspace.SnapshotMetadata*
  : Given a tag, return the asssociated snapshot metadata. This lookup could be slower ,if a reverse index is not kept.

get_snapshot_by_tag_or_hash(*tag_or_hash: str*) → *dataworkspaces.workspace.SnapshotMetadata*
Given a string that is either a tag or a (partial)hash corresponding to a snapshot, return the associated resrouce metadata. Throws a ConfigurationError if no entry is found.

get_snapshot_manifest(*hash_val: str*) → List[Any]
Returns the snapshot manifest for the given hash as a parsed JSON structure. The top-level dict maps resource names resource parameters.

abstract **get_snapshot_metadata**(*hash_val: str*) → *dataworkspaces.workspace.SnapshotMetadata*
Given the full hash of a snapshot, return the metadata. This lookup should be quick.

abstract **list_snapshots**(*reverse: bool = True*, *max_count: Optional[int] = None*) → Iterable[*dataworkspaces.workspace.SnapshotMetadata*]
Returns an iterable of snapshot metadata, sorted by timestamp ascending (or descending if reverse is True). If max_count is specified, return at most that many snaphsots.

abstract **remove_tag_from_snapshot**(*hash_val: str*, *tag: str*) → None
Remove the specified tag from the specified snapshot. Throw an InternalError if either the snapshot or the tag do not exist.

restore(*snapshot_hash: str*, *restore_hashes: Dict[str, Optional[str]]*, *restore_resources: List[*dataworkspaces.workspace.SnapshotResourceMixin*]*) → None
Restore the specified resources to the specified hashes. The list should have been previously filtered to include only those with valid (not None) restore hashes.

abstract **save_snapshot_metadata_and_manifest**(*metadata: dataworkspaces.workspace.SnapshotMetadata*, *manifest: bytes*) → None
Save the snapshot metadata and manifest using the hash in metadata.hashval.

snapshot(*tag: Optional[str] = None*, *message: str = ''*) → Tuple[*dataworkspaces.workspace.SnapshotMetadata*, bytes]
Take snapshot of the resources in the workspace, and metadata for the snapshot and a manifest in the workspace. We assume that the tag does not already exist (checks can be made in the command before calling this method).

We also copy the lineage data if the workspace supports lineage.

Returns the snapshot metadata and the (binary) snapshot hash. These should be saved into the workspace by the caller (i.e. the snapshot command). We don't do that here, as futher interactions with the user may be needed. In particular, if the hash is identical to a previous hash, we ask the user if they want to overwrite.

abstract **supports_lineage**() → bool
Return True if this workspace's backend supports lineage, False otherwise

write_export_lineage_for_snapshot(*current_resources: List[*dataworkspaces.workspace.Resource*]*) → None
For all exported resources, we write out the lineage.json file in the root directoryfor the resource.

write_result_lineage_for_snapshot(*current_resources: List[*dataworkspaces.workspace.Resource*]*, *rel_dest_path: str*) → None
For all results resources, we write out the lineage.json files in the snapshot directory.

class dataworkspaces.workspace.**SnapshotResourceMixin**
Mixin for the resource api for resources that can take snapshots.

copy_imported_lineage(*lineage_store: dataworkspaces.utils.lineage_utils.LineageStore*) → None
If imported lineage, copy the lineage.json file to the lineage store. The pull_resources() method on the workspace will call it after pulling the resource.

If the resource does not store files locally, this default implementation will need to be overridden.

abstract delete_snapshot(*workspace_snapshot_hash: str*, *resource_restore_hash: str*, *relative_path: str*) → None

Delete any state associated with the snapshot, including any files under relative_path

abstract restore(*restore_hashval: str*) → None

abstract restore_precheck(*restore_hashval: str*) → None

Run any prechecks before restoring to the specified hash value (aka certificate). This should throw a ConfigurationError if the restore would fail for some reason.

abstract snapshot() → Tuple[Optional[str], Optional[str]]

Take the actual snapshot of the resource and return a tuple of two hash values, the first for comparison, and the second for restoring. The comparison hash value is the one we save in the snapshot manifest. The restore hash value is saved in the snapshot metadata. In many cases both hashes are the same. If the resource does not support restores, it can return None for the second hash. This will cause attempted restores involving this resource to error out.

abstract snapshot_precheck() → None

Run any prechecks before taking a snapshot. This should throw a ConfigurationError if the snapshot would fail for some reason.

1.8 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX