SDK Reference v26.1.1

See all our docs at docs.snorkel.ai

PDF generated on March 4, 2026

SDK overview

The Snorkel SDK provides programmatic access to the Snorkel AI Data Development Platform, enabling you to build, deploy, and manage data-centric AI applications at scale.

Get started

New to the Snorkel SDK? Start here:

- **SDK quickstart** - Get up and running with the SDK in minutes
- **Notebook tips** - Best practices and helpful tips for using the SDK in Jupyter notebooks

Main SDK components

The Snorkel SDK is organized into three main modules:

- snorkelai.sdk.client: Interfaces to interact with Snorkel's REST API.
- snorkelai.sdk.develop: Object-oriented SDK for Snorkel.
- snorkelai.sdk.utils: Utilities for working with Snorkel.

API reference

For detailed API documentation, see the API Reference section which contains comprehensive documentation for all SDK classes, methods, and functions.

SDK quickstart

This guide shows you how to access or install Snorkel's SDK, and then use it to explore core data management functions.

- Install the SDK locally
- Access the SDK from a Snorkel-hosted instance

Install the SDK locally

Requirements

- Python version 3.10-3.13
- A Snorkel account with admin or developer access
- Your Snorkel API key
- Your Snorkel instance host name or IP address. You can copy the host name from your Snorkel instance's URL.

> **TIP**
>
> Use a Conda environment for your local Snorkel installation. For example, you can run these commands to create and activate a Python 3.11 environment:
>
> ```
> conda create -n snorkel-env python=3.11
>
> conda activate snorkel-env
> ```

Install the SDK

Run the following command to install the basic Snorkel SDK in your Python environment:

```
pip install snorkelai-sdk
```

Authentication

All API requests to Snorkel must be authenticated. You need an API key, and the Python SDK client needs to use an authenticated `SnorkelSDKContext` object to make API requests to Snorkel services.

Follow these instructions to generate an API key:

- User and Admin settings

When you connect to Snorkel locally or from another external system, you must provide additional settings and authentication secrets. Use the following connection template:

```
import os

# Core Snorkel SDK imports
import snorkelai.sdk.client as sai
from snorkelai.sdk.develop import Dataset, Slice, Batch

print("✅ Snorkel AI SDK imported successfully")

# Snorkel SDK configuration
SAI_CONFIG = {
   "endpoint": "https://<your-snorkel-hostname>",
   "api_key": "<your-api-key>",
   "workspace_name": "<your-workspace-name>",
   "debug": True  # Optional: set to False to disable debug logging
}

# Initialize Snorkel context
ctx = sai.SnorkelSDKContext.from_endpoint_url(**SAI_CONFIG)
```

- `endpoint`: You can copy the host name from your Snorkel instance's URL.
- `api_key`: Your Snorkel API key

Access the SDK from a Snorkel-hosted instance

Requirements

- When you deploy Snorkel, you must enable in-platform notebook support. Change the following flag in your Helm `values.yaml` file:

Name	Value
services.jupyterhub	`{"enabled": true}`

- A Snorkel account with admin or developer access

Access the SDK

You can call the Snorkel SDK from a Jupyter notebook hosted on your Snorkel instance.

1. Select **Notebook** from the left navigation.
2. Select **+** to create a new notebook from scratch, or upload one.
3. Select **Python 3 (ipykernel)** from the **Notebook** section. A new Jupyter notebook opens.

Authentication

When using a Snorkel-hosted notebook, Snorkel automatically generates an API key for your user and assigns it as an environment variable. When you log out and log back in, Snorkel rotates this key to ensure security.

```
import snorkelai.sdk.client as sai

# Set your workspace
workspace_name = 'YOUR_WORKSPACE_NAME' #INPUT - Replace with your workspace
name

# Configure client context for Snorkel instance
ctx = sai.SnorkelSDKContext.from_endpoint_url(
    workspace_name=workspace_name,
)
```

Note that the `workspace_name` is not required when using the default workspace.

Now you can use Snorkel's SDK.

Quickstart: SDK connection and dataset exploration

This quickstart guide will walk you through connecting to Snorkel and exploring your workspace using the SDK. You'll learn how to authenticate and explore core data management functions.

Getting started

Begin by importing the necessary packages and setting up your connection:

```
import snorkelai.sdk.client as sai
from snorkelai.sdk.develop import Dataset, Slice

# Connect to Snorkel using the authentication block from above
# ctx = sai.SnorkelSDKContext.from_endpoint_url( ... ) # Input external or
Snorkel-hosted connection details

print("✅ SDK imported successfully")
```

For the connection, uncomment the `ctx` line and replace it with the full authentication block from the appropriate section above.

Explore your workspace

Explore datasets available in your workspace:

```
# Verify connection
print(f"Connected to workspace: {ctx.workspace_name}")

# List existing datasets in your workspace
# Note: For workspaces with many datasets, this may take a moment
print("\nRetrieving datasets from workspace...")
datasets = Dataset.list()
print(f"Found {len(datasets)} datasets in workspace")

# Show first few datasets only to avoid overwhelming output
max_display = 5
datasets_to_show = datasets[:max_display]
print(f"\nShowing first {len(datasets_to_show)} datasets:")

for dataset in datasets_to_show:
    print(f"- {dataset.name}")
    print(f"  UID: {dataset.uid}")
    print(f"  Created: {getattr(dataset, 'created_at', 'Unknown')}")
    print()

if len(datasets) > max_display:
    print(f"... and {len(datasets) - max_display} more datasets")
```

Create a dataset

Create a new dataset to demonstrate SDK capabilities:

```
# Create a new dataset (metadata only - no data upload required)
import datetime
timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
dataset_name = f"quickstart-demo-{timestamp}"

new_dataset = Dataset.create(dataset_name)
print(f"✅ Created dataset: {new_dataset.name}")
print(f"Dataset UID: {new_dataset.uid}")
```

Display dataset metadata

Explore dataset information without accessing the underlying data:

```
# Refresh dataset list to include our new dataset
datasets = Dataset.list()
print(f"\nNow have {len(datasets)} datasets in workspace")

# Show our newly created dataset and a couple others
print(f"\nRecent datasets (showing up to 3):")
recent_datasets = datasets[:3]  # Limit to first 3 for display
for dataset in recent_datasets:
    marker = "← NEW" if dataset.name == new_dataset.name else ""
    print(f"- {dataset.name} {marker}")
    print(f"  UID: {dataset.uid}")

if datasets:
    # Work with our newly created dataset
    dataset = new_dataset
    print(f"\nExploring our new dataset: {dataset.name}")
    print(f"Dataset UID: {dataset.uid}")

    # Show available dataset properties
    print(f"\nDataset properties:")
    for attr in ['name', 'uid', 'created_at', 'description']:
        if hasattr(dataset, attr):
            value = getattr(dataset, attr)
            print(f"- {attr}: {value}")

    print("✅ Dataset metadata exploration successful")
else:
    print("No datasets found.")
```

Explore dataset organization with slices

View how data is organized with slices:

```
if datasets:
    # Use the first existing dataset (not our empty new one)
    dataset = datasets[0] if datasets[0].name != new_dataset.name else
(datasets[1] if len(datasets) > 1 else datasets[0])

    # List existing slices for the dataset
    existing_slices = Slice.list(dataset=dataset.uid)
    print(f"\nSlices for dataset '{dataset.name}':")
    print(f"Found {len(existing_slices)} slices:")

    for slice_obj in existing_slices:
        print(f"- {slice_obj.name}")
        print(f"  Description: {slice_obj.description}")
        # Note: Some slice objects may not have uid attribute
        if hasattr(slice_obj, 'uid'):
            print(f"  UID: {slice_obj.uid}")
        print()

    print("✅ Slice exploration complete")
```

SDK method verification

Finally, verify key SDK functions:

```
print("\n=== SDK Method Verification ===")

# Test context properties
print(f"Workspace access: {'✅ Available' if hasattr(ctx, 'workspace_name')
else '❌ Missing'}")
print(f"Debug mode: {'✅ Available' if hasattr(ctx, 'set_debug') else '❌
Missing'}")

# Test Dataset class methods
print(f"Dataset listing: {'✅ Available' if hasattr(Dataset, 'list') else '❌
Missing'}")
print(f"Dataset creation: {'✅ Available' if hasattr(Dataset, 'create') else
'❌ Missing'}")

# Test Slice class methods
print(f"Slice listing: {'✅ Available' if hasattr(Slice, 'list') else '❌
Missing'}")
print(f"Slice creation: {'✅ Available' if hasattr(Slice, 'create') else '❌
Missing'}")

print(f"\nWorkspace summary:")
print(f"- Workspace: {ctx.workspace_name}")
print(f"- Total datasets: {len(datasets) if 'datasets' in locals() else 0}")
print(f"- SDK connection: ✅ Working")

print("\n✅ SDK exploration complete!")
```

In this tutorial, you've successfully connected to the Snorkel AI Data Development Platform and explored your workspace using the SDK. You've seen how to authenticate, list datasets, explore data organization, and verify SDK functionality. This provides the foundation for more advanced SDK workflows.

Next steps

Explore the SDK reference documentation.

Notebook tips

This document contains a collection of tips and best practices for using Jupyter notebooks effectively with Snorkel.

How to use Git CLI with Snorkel

Snorkel's in-platform notebook server comes with Git CLI, which allows users to version control notebooks, data files, and more.

1. Select **Notebook** from the left navigation.
2. From the **Other** menu, select **Terminal**.
3. Clone your git repository using the full HTTPS URL, like `https://github.com/USERNAME/REPO.git`:

   ```
   $ git clone https://github.com/USERNAME/REPO.git
   Username: YOUR_USERNAME
   Password: YOUR_PERSONAL_ACCESS_TOKEN
   ```

   Use your personal access token. See Using a personal access token on the command line for details.

> NOTE
>
> Note that SSH URLs, like `git@github.com:user/repo.git`, are not supported.

> NOTE
>
> Snorkel does not come with its own internal Git repository. Please create or provide your own from GitHub or equivalent.

The Git CLI is also available through the Notebook via system shell access (i.e., the magic command `!git`).

API Reference

`snorkelai.sdk.develop`	Object-oriented interfaces of Snorkel SDK
`snorkelai.sdk.client`	Interfaces to interact with the Snorkel REST API.
`snorkelai.sdk.utils`	Other general utilities.

snorkelai.sdk.client

Interfaces to interact with the Snorkel REST API.

Most of the functions in the `snorkelai.sdk.client` module require a *client context* object — `SnorkelSDKContext` — that points to the Snorkel Flow instance:

```
import snorkelai.sdk.client as sai
ctx = sai.SnorkelSDKContext.from_endpoint_url()
```

All the functions under submodules are also available under `snorkelai.sdk.client`.

Examples

```
import snorkelai.sdk.client as sai
# get_annotation_sources is available under
snorkelai.sdk.client.annotation_sources
sai.annotation_sources.get_annotation_sources()
# also available under snorkelai.sdk.client (recommended)
sai.get_annotation_sources()  # noqa: F405
```

Since `snorkelai.sdk.client` submodules may be reorganized in the future, we recommend accessing functions directly from `snorkelai.sdk.client` to minimize the risk of future breaking changes.

Submodules

`snorkelai.sdk.client.annotation_sources`	Annotation source related functions.
`snorkelai.sdk.client.connector_configs`	Data connector related functions
`snorkelai.sdk.client.ctx`	Context related classes.
`snorkelai.sdk.client.external_models`	External model endpoints related functions.
`snorkelai.sdk.client.files`	File storage related functions to upload and download files and directories.
`snorkelai.sdk.client.fm_suite`	Foundation model suite related functions.

`snorkelai.sdk.client.secrets`	Secret store related functions.
`snorkelai.sdk.client.synthetic`	Synthetic data related functions for generating synthetic data.
`snorkelai.sdk.client.utils`	Utility functions.
`snorkelai.sdk.client.users`	User related functions.

snorkelai.sdk.client.annotation_sources

Annotation source related functions.

Functions

`create_annotation_source`([username, ...])	Create an annotation source.
`delete_annotation_source`(source_name)	Delete an annotation source.
`get_annotation_source`([source_name, source_uid])	Get an annotation source from uid or name.
`get_annotation_sources`()	Get annotation sources.
`update_annotation_source`(source_name[, ...])	Update an annotation source.

snorkelai.sdk.client.annotation_sources.create_annotation_source

snorkelai.sdk.client.annotation_sources.create_annotation_source(*username=None, source_name=None, source_type=None, metadata=None*)

Create an annotation source.

Parameters

Name	Type	Default	Info
username	`Optional[str]`	`None`	The username of source if the source type is user.
source_name	`Optional[str]`	`None`	The name of the source, which defaults to username for user type of source.
source_type	`Optional[str]`	`None`	The type of source (user, aggregation, etc).
metadata	`Optional[dict]`	`None`	Any source metadata.

Returns

The created source.

Return type

`Source`

snorkelai.sdk.client.annotation_sources.delete_annotation_source

snorkelai.sdk.client.annotation_sources.delete_annotation_source(*source_name*)

Delete an annotation source.

Parameters

Name	Type	Default	Info
source_name	`str`		The name of the source.

Returns

Returns true if the operation succeeds.

Return type

`bool`

snorkelai.sdk.client.annotation_sources.get_annotation_source

snorkelai.sdk.client.annotation_sources.get_annotation_source(*source_name=None, source_uid=None*)

Get an annotation source from uid or name.

Parameters

Name	Type	Default	Info
source_name	`Optional[str]`	`None`	Name of the source (such as username, etc).
source_uid	`Optional[int]`	`None`	Uid of the source.

Return type

`A single source that matches name and/or uid`

snorkelai.sdk.client.annotation_sources.get_annotation_sources

snorkelai.sdk.client.annotation_sources.get_annotation_sources()

Get annotation sources.

Returns

A list of sources.

Return type

`List[Source]`

snorkelai.sdk.client.annotation_sources.update_annotation_source

snorkelai.sdk.client.annotation_sources.update_annotation_source(*source_name*, *new_source_name=None*, *metadata=None*)

Update an annotation source.

Parameters

Name	Type	Default	Info
source_name	`str`		The current name of the source.
new_source_name	`Optional[str]`	`None`	The new name of the source.
metadata	`Optional[dict]`	`None`	The updated metadata.

Returns

The updated source.

Return type

`Source`

snorkelai.sdk.client.connector_configs

Data connector related functions

Functions

`create_connector_config`(name, ...)	Create a new connector configuration.
`delete_connector_config`(config_uid)	Delete a connector configuration.
`get_connector_config`(config_uid)	Get a connector configuration by its UID.
`list_connector_configs`(connector_type)	List all configurations for a specific connector type.
`update_connector_config`(config_uid, ...)	Update a connector configuration.

snorkelai.sdk.client.connector_configs.create_connector_config

snorkelai.sdk.client.connector_configs.create_connector_config(*name, connector_type, config*)

Create a new connector configuration.

Parameters

Name	Type	Default	Info
name	`str`		The name of the connector configuration.
connector_type	`str`		The type of connector to create a configuration (e.g. “AmazonS3”).
config	`Dict[str, Any]`		The connector configuration.

Returns

The UID of the created connector configuration

Return type

`int`

Examples

```
>>> from snorkelai.sdk.client import create_connector_config
>>> config_uid = create_connector_config(
...     name="my_s3_config",
...     connector_type="AmazonS3",
...     config={
...         "access_key_id": "my_access_key_id",
...         "secret_access_key": "my_secret_access_key",
...         "region": "us-east-1",
...     },
... )
>>> print(config_uid)
```

snorkelai.sdk.client.connector_configs.delete_connector_config

snorkelai.sdk.client.connector_configs.delete_connector_config(*config_uid*)

Delete a connector configuration.

Parameters

Name	Type	Default	Info
config_uid	`int`		The UID of the connector configuration to delete.

Return type

`None`

Examples

```
>>> from snorkelai.sdk.client import delete_connector_config
>>> delete_connector_config(123)
```

snorkelai.sdk.client.connector_configs.get_connector_config

snorkelai.sdk.client.connector_configs.get_connector_config(*config_uid*)

Get a connector configuration by its UID.

Parameters

Name	Type	Default	Info
config_uid	`int`		The UID of the connector configuration to get.

Returns

A dictionary representing the connector configuration

Return type

`Dict[str, Any]`

Examples

```
>>> from snorkelai.sdk.client import get_connector_config
>>> config = get_connector_config(123)
>>> print(config)
```

snorkelai.sdk.client.connector_configs.list_connector_configs

snorkelai.sdk.client.connector_configs.list_connector_configs(*connector_type*)

List all configurations for a specific connector type.

Parameters

Name	Type	Default	Info
connector_type	`str`		The type of connector to list configurations for (e.g. “AmazonS3”).

Returns

A list of connector configurations of the given type

Return type

`List[Dict[str, Any]]`

Examples

```
>>> from snorkelai.sdk.client import list_connector_configs
>>> configs = list_connector_configs("AmazonS3")
>>> print(configs)
```

snorkelai.sdk.client.connector_configs.update_connector_config

snorkelai.sdk.client.connector_configs.update_connector_config(*config_uid, new_name, new_config*)

Update a connector configuration.

Parameters

Name	Type	Default	Info
config_uid	`int`		The UID of the connector configuration to update.
new_name	`str`		The new name of the connector configuration.
new_config	`Dict[str, Any]`		The new configuration for the connector.

Return type

`None`

Examples

```
>>> from snorkelai.sdk.client import update_connector_config
>>> new_config = {
...     "access_key_id": "new_access_key_id",
...     "secret_access_key": "new_secret_access_key",
...     "region": "new_region",
... }
>>> update_connector_config(123, "new_name", new_config)
```

snorkelai.sdk.client.ctx.SnorkelSDKContext

class **snorkelai.sdk.client.ctx.SnorkelSDKContext**(*tdm_client, storage_client=None, workspace_name=None, set_global=True, debug=False*)

Bases: `object`

The SnorkelSDKContext object provides client context for the Snorkel Flow SDK. It allows the Snorkel Flow SDK to recognize a Snorkel Flow instance by identifying Snorkel Flow's essential API services (TDM & Storage API) via user-provided parameters, a YAML config file, or a Snorkel Flow API key.

Examples

```
import snorkelai.sdk.client as sai
ctx = sai.SnorkelSDKContext.from_endpoint_url(...)
```

__init__

__init__(*tdm_client, storage_client=None, workspace_name=None, set_global=True, debug=False*)

Initialize a SnorkelSDKContext.

Methods

`__init__`(tdm_client[, storage_client, ...])	Initialize a SnorkelSDKContext.
`from_endpoint_url`([endpoint, api_key, ...])	Initialize a SnorkelSDKContext from keyword arguments.
`get_global`()	Retrieve the global SnorkelSDKContext object.
`set_debug`(debug)	Set the verbosity of warnings, details, and stacktraces
`set_global`([ctx])	Set a SnorkelSDKContext object globally.

Attributes

`storage_client`	
`workspace_name`	SnorkelSDKContext objects are only scoped to work in a particular workspace.

from_endpoint_url

classmethod **from_endpoint_url**(*endpoint=None, api_key=None, workspace_name=None, set_global=True, debug=False, *args, **kwargs*)

Initialize a SnorkelSDKContext from keyword arguments.

Examples

```
# Instantiate with default kwargs and a custom url
import snorkelai.sdk.client as sai
ctx =
sai.SnorkelSDKContext.from_endpoint_url("https://edge.k8s.g498.io/")
```

```
# Instantiate with custom kwargs
import snorkelai.sdk.client as sai
ctx = sai.SnorkelSDKContext.from_endpoint_url(
    endpoint="https://edge.k8s.g498.io/",
    workspace_name="my-workspace",
)
```

```
# Instantiate with an API key
import snorkelai.sdk.client as sai
ctx = sai.SnorkelSDKContext.from_endpoint_url(
    endpoint="https://edge.k8s.g498.io/",
    api_key="my-api-key",
)
```

Parameters

Name	Type	Default	Info
endpoint	`Optional[str]`	`None`	The baseurl to a snorkelflow instance.
workspace_name	`Optional[str]`	`None`	The workspace name, which determines the workspace a dataset and application will be created in and queried from.

api_key	`Optional[str]`	`None`	The API key to use for the TDM and Storage API.
set_global	`bool`	`True`	Whether to set the context as the global context, accessible across all notebooks.
debug	`bool`	`False`	Whether to enable debug mode. Debug mode will print more verbose errors to the screen inside of the Python notebook.

Returns

A SnorkelSDKContext object that can be used to interact with the TDM and Storage API clients directly.

Return type

SnorkelSDKContext

get_global

classmethod **get_global**()

Retrieve the global SnorkelSDKContext object.

Examples

```
import snorkelai.sdk.client as sai
ctx = sai.SnorkelSDKContext.get_global()
```

Returns

A SnorkelSDKContext object that can be used to interact with the TDM and Storage API clients directly.

Return type

SnorkelSDKContext

Raises

AttributeError – If no global context has been set.

set_debug

set_debug(*debug*)

Set the verbosity of warnings, details, and stacktraces

Parameters

Name	Type	Default	Info
debug	`bool`		If True, SDK operations will print more verbose errors to the screen inside of the Python notebook.

Return type

`None`

set_global

classmethod **set_global**(*ctx=None*)

Set a SnorkelSDKContext object globally. This context object will be used by all Snorkel Platform SDK functions.

Examples

```
import snorkelai.sdk.client as sai
ctx = sai.SnorkelSDKContext.from_endpoint_url(...)
sai.SnorkelSDKContext.set_global(ctx)
```

Parameters

Name	Type	Default	Info
ctx	`Optional[SnorkelSDKContext]`	`None`	A context to set as global. If no context is provided, will attempt to create one with default parameters.

Return type

`None`

property **storage_client***: StorageClient*

property **workspace_name***: str*

SnorkelSDKContext objects are only scoped to work in a particular workspace.

Returns

The name of the workspace belonging to this context object.

Return type

`str`

snorkelai.sdk.client.external_models

External model endpoints related functions. External models endpoints are used to configure specific models from foundation model providers.

Functions

`delete_external_model_endpoint`(model_name)	Removes an external model endpoint from DB (Only for superadmin users).
`get_external_model_endpoints`([model_name, ...])	Gets the external model endpoints from DB (Only for superadmin users).
`set_external_model_endpoint`(model_name, ...)	Adds an external model endpoint to DB (Only for superadmin users).

snorkelai.sdk.client.external_models.delete_external_model_endpoint

snorkelai.sdk.client.external_models.delete_external_model_endpoint(*model_name*)

Removes an external model endpoint from DB (Only for superadmin users).

Parameters

Name	Type	Default	Info
model_name	`str`		Name of the model.

Return type

`None`

snorkelai.sdk.client.external_models.get_external_model_endpoints

snorkelai.sdk.client.external_models.get_external_model_endpoints(*model_name=None, detail=False*)

Gets the external model endpoints from DB (Only for superadmin users).

Parameters

Name	Type	Default	Info
model_name	`Optional[str]`	`None`	Name of the model to retrieve the endpoint or configuration for. Defaults to None to return specified information for all models.
detail	`bool`	`False`	Whether to return the full configuration for each model. Defaults to False to return only the endpoint URL.

Returns

Mapping of model name to model configuration

Return type

`Dict[str, Any]`

snorkelai.sdk.client.external_models.set_external_model_endpoint

snorkelai.sdk.client.external_models.set_external_model_endpoint(*model_name, endpoint, model_provider, fm_type, **config_kwargs*)

Adds an external model endpoint to DB (Only for superadmin users). NOTE: this will impact all users who elect to use *model_name*

Parameters

Name	Type	Default	Info
model_name	`str`		Name of the model.
endpoint	`str`		Endpoint of the model.
model_provider	`str`		Name of the model provider (one of: azure_ml, azure_openai, bedrock, custom_inference_service, huggingface, openai, vertexai_lm).
fm_type	`str`		The model type of the foundation mode (one of: text2text, qa, docvqa).
config_kwargs	`Any`		Any additional config options to set for the model.

Return type

`None`

snorkelai.sdk.client.files

File storage related functions to upload and download files and directories.

Functions

`download_dir`(remote_path, local_path)	Downloads remote directory from Snorkel Files Store to local directory.
`download_file`(remote_path, local_path)	Downloads remote file from Snorkel Files Store to local file.
`list_dir`(remote_path)	Lists files in remote directory from Snorkel Files Store.
`upload_dir`(local_path, remote_path)	Uploads a local directory to the Snorkel Files Store.
`upload_file`(local_path, remote_path)	Uploads a file to the Snorkel Files Store.

snorkelai.sdk.client.files.download_dir

snorkelai.sdk.client.files.download_dir(*remote_path, local_path*)

Downloads remote directory from Snorkel Files Store to local directory. Files and subdirectories inside the remote directory will be placed directly in the local directory. Both absolute paths (ex. minio://workspace-1/data_dir) and relative paths (ex. data_dir, workspace-1/data_dir) are supported.

⚠ WARNING

If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/data_dir -> data_dir.

Example

```
remote_path = "minio://workspace-1/data_dir" # equivalent to "data_dir"
local_path = "/home/user/download_dir"

# Files and sub-directories under `remote_path` will be downloaded to
/home/user/download_dir
sai.download_dir(remote_path, local_path)

# To preserve the directory name, you can specify the local path like
this:
sai.download_dir(remote_path, "/home/user/download_dir/data_dir")

# These calls will raise a ValueError
sai.download_dir("minio://workspace-{not-current-workspace-
id}/data_dir", local_path)
sai.download_dir("workspace-{not-current-workspace-id}/data_dir",
local_path)
```

Parameters

Name	Type	Default	Info
remote_path	`str`		Path of remote directory to be downloaded.
local_path	`str`		Local directory to download file to.

Return type

`None`

snorkelai.sdk.client.files.download_file

snorkelai.sdk.client.files.download_file(*remote_path, local_path*)

Downloads remote file from Snorkel Files Store to local file. Both absolute paths (ex. minio://workspace-1/data/file.txt) and relative paths (ex. data/file.txt, workspace-1/data/file.txt) are supported.

> ⚠ WARNING
>
> If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/file.txt -> file.txt.

Example

```
remote_path = "minio://workspace-1/data/file.txt"
local_path = "/home/user/file.txt"

# File will be downloaded to /home/user/file.txt
sai.download_file(remote_path, local_path)

# These calls will raise a ValueError
sai.download_file("minio://workspace-{not-current-workspace-
id}/data/file.txt", local_path)
sai.download_file("workspace-{not-current-workspace-id}/data/file.txt",
local_path)
```

Parameters

Name	Type	Default	Info
remote_path	`str`		Path of file to be downloaded.
local_path	`str`		Local path to download file to.

Return type

`None`

snorkelai.sdk.client.files.list_dir

snorkelai.sdk.client.files.list_dir(*remote_path*)

Lists files in remote directory from Snorkel Files Store. Both absolute paths (ex. minio://workspace-1/data_dir) and relative paths (ex. data_dir, workspace-1/data_dir) are supported.

⚠ WARNING

If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/data_dir -> data_dir.

Example

```
remote_path = "minio://workspace-1/data_dir" # equivalent to "data_dir"

# Files in the remote directory will be listed
sai.list_dir(remote_path)

# These calls will raise a ValueError
sai.list_dir("minio://{not-current-workspace-id}/data_dir")
sai.list_dir("{not-current-workspace-id}/data_dir")
```

Parameters

Name	Type	Default	Info
remote_path	`str`		Path to directory to be listed.

Returns

List of files in specified remote directory

Return type

`List[Any]`

snorkelai.sdk.client.files.upload_dir

snorkelai.sdk.client.files.upload_dir(*local_path*, *remote_path*)

Uploads a local directory to the Snorkel Files Store. Both absolute paths (ex. minio://workspace-1/data_dir) and relative paths (ex. data_dir, workspace-1/data_dir) are supported.

WARNING

If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/data_dir -> data_dir.

Example

```
local_path = "/home/user/data_dir"
remote_path = "data_dir"

# Directory will be uploaded to minio://workspace-{current-workspace-
id}/data_dir
sai.upload_dir(local_path, remote_path)

# These calls will raise a ValueError
sai.upload_dir(local_path, "minio://workspace-{not-current-workspace-
id}/data_dir")
sai.upload_dir(local_path, "workspace-{not-current-workspace-
id}/data_dir")
```

Parameters

Name	Type	Default	Info
local_path	`str`		Directory containing files to be uploaded.
remote_path	`str`		Remote directory on Snorkel Files Store to upload files to.

Returns

Tuple containing:

- List of uploaded file paths
- Uploaded directory path

Return type

`Tuple[List[str], str]`

snorkelai.sdk.client.files.upload_file

snorkelai.sdk.client.files.upload_file(*local_path, remote_path*)

Uploads a file to the Snorkel Files Store. Both absolute paths (ex. minio://workspace-1/file.txt) and relative paths (ex. file.txt, workspace-1/file.txt) are supported.

> ⚠ WARNING
>
> If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/file.txt -> file.txt.

Example

```
local_path = "/home/user/file.txt"
remote_path = "file.txt"

# File will be uploaded to minio://workspace-{current-workspace-
id}/file.txt
sai.upload_file(local_path, remote_path)

# These calls will raise a ValueError
sai.upload_file(local_path, "minio://workspace-{not-current-workspace-
id}/file.txt")
sai.upload_file(local_path, "workspace-{not-current-workspace-
id}/file.txt")
```

Parameters

Name	Type	Default	Info
local_path	`str`		Path to file to be uploaded.
remote_path	`str`		File path in Snorkel Files Store to upload file to.

Returns

The uploaded file path

Return type

`str`

snorkelai.sdk.client.fm_suite

Foundation model suite related functions. Provides functions for estimating costs, viewing available methods, running jobs, and monitoring progress.

Functions

`prompt_fm`(prompt, model_name[, model_type, ...])	Send one or more prompts to a foundation model
`prompt_fm_over_dataset`(prompt_template, ...)	Run a prompt over a dataset.

snorkelai.sdk.client.fm_suite.prompt_fm

snorkelai.sdk.client.fm_suite.prompt_fm(*prompt, model_name, model_type=None, question=None, runs_per_prompt=1, sync=True, cache_name='default', **fm_hyperparameters*)

Send one or more prompts to a foundation model

Parameters

Name	Type	Default	Info
prompt	`Union[str, List[str]]`		The prompt(s) to send to the foundation model.
model_name	`str`		The name of the foundation model to use.
model_type	`Optional[LLMType]`	`None`	The way we should use the foundation model, must be one of the LLMType values.
question	`Optional[str]`	`None`	When provided, this get's passed to the model for each prompt which is useful for information retrieval tasks. The prompt argument essentially then becomes the context(s) which contains the answer to the question.

runs_per_prompt	`int`	`1`	The number of times to run a prompt, note each response can be different. All will be cached.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
cache_name	`str`	`'default'`	The cache name is used in the hash construction. To run a prompt and get a different result, you should change the cache name to something that hasn't been used before. For example: >> sai.prompt_fm("What is the meaning of life?", "openai/gpt-4o") The meaning of life is to work… >> sai.prompt_fm("What is the meaning of life?", "openai/gpt-4o") << hit's the cache The meaning of life is to work… >> sai.prompt_fm("What is the meaning of life?", "openai/gpt-4o", cache_name="run_2")

			<< hit's a different part of the cache The meaning of life is to have fun!.
fm_hyperparameters	`Any`		Additional keyword arguments to pass to the foundation model such as temperature, max_tokens, etc.

Return type

`Union`[`DataFrame`, `str`]

Returns

- *df* – Dataframe containing the predictions for the data points. There are two columns, the input prompt and the output of the foundation model.
- *job_id* – The job id of the prompt inference job which can be used to monitor progress with sai.poll_job_status(job_id).

Examples

```
>>> sai.prompt_fm(prompt="What is the meaning of life?",
model_name="openai/gpt-4")
   | text                           | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | What is the meaning of life?   | Life is all about having fun!
| 0.789
```

```
>>> sai.prompt_fm(prompt=["What is the meaning of life?", "What is the
meaning of death?"], model_name="openai/gpt-4")
   | text                           | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | What is the meaning of life?   | Life is all about having fun!
| 0.789
1  | What is the meaning of death?  | Death is about not having fun!
| 0.981
```

```
>>> sai.prompt_fm(question="What is surname", prompt="Joe Bloggs is a
person", model_name="deepset/roberta-base-squad2")
   | text                           | answer          | start  | end   |
score
------------------------------------------------------------------------
-------------------------------------
0  | Joe Bloggs is a person         | Bloggs          | 4      | 11    |
0.985
```

snorkelai.sdk.client.fm_suite.prompt_fm_over_dataset

snorkelai.sdk.client.fm_suite.prompt_fm_over_dataset(*prompt_template, dataset, x_uids, model_name, model_type=None, runs_per_prompt=1, sync=True, cache_name='default', system_prompt=None, **fm_hyperparameters*)

Run a prompt over a dataset. Any field in the dataset can be referenced in the prompt by using curly braces, {field_name}.

Parameters

Name	Type	Default	Info
prompt_template	`str`		The prompt template used to format input rows sent to the foundation model.
dataset	`Union[str, int]`		The name or UID of the dataset containing the data we want to prompt over.
x_uids	`List[str]`		The x_uids of the rows within the dataset to prompt over.
model_name	`str`		The name of the foundation model to use.
model_type	`Optional[LLMType]`	`None`	The way we should use the foundation model, must be one of the LLMType values.

runs_per_prompt	`int`	`1`	The number of times to run inference over an xuid, note each response can be different. All will be cached.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
cache_name	`str`	`'default'`	The cache name is used in the hash construction. To run a prompt and get a different result, you should change the cache name to something that hasn't been used before. For example: >> sai.prompt_fm("What is the meaning of life?", "openai/gpt-4o") The meaning of life is to work… >> sai.prompt_fm("What is the meaning of life?", "openai/gpt-4o") << hit's the cache The meaning of life is to work… >> sai.prompt_fm("What is the meaning of life?", "openai/gpt-4o",

			cache_name=”run_2”) << hit’s a different part of the cache The meaning of life is to have fun!.
system_prompt	`Optional[str]`	`None`	The system prompt to prepend to the prompt.
fm_hyperparameters	`Any`		Additional keyword arguments to pass to the foundation model such as temperature, max_tokens, etc.

Return type

`Union`[`DataFrame`, `str`]

Returns

- *df* – Dataframe containing the predictions for the data points. There are two columns, the input prompt and the output of the foundation model.
- *job_id* – The job id of the prompt inference job which can be used to monitor progress with sai.poll_job_status(job_id).

Examples

```
>>> sai.prompt_fm_over_dataset(prompt_template="{email_subject}. What is
this email about?", dataset=1, x_uids=["0", "1"],
model_name="openai/gpt-4")
   | email_subject                          | generated_text
| perplexity
------------------------------------------------------------------------
-----------------------------------------------------------
0  | Fill in survey for $50 amazon voucher  | The email is asking you to
fill in a survey for an amazon voucher       | 0.891
1  | Hey it's Bob, free on Sat?             | The email is from your
friend Bob as if you're free on Saturday         | 0.787
```

snorkelai.sdk.client.secrets

Secret store related functions.

Functions

`delete_secret`(key[, secret_store, ...])	Deletes a secret from the secret store (Only for superadmin users).
`get_model_provider_status`(model_provider)	Gets the status of a model provider.
`list_integrations`([secret_store, ...])	Gets configuration status for each foundation model provider (Only for superadmin users).
`list_secrets`([secret_store, workspace_uid, ...])	Gets all secret keys in a workspace (Only for superadmin users).
`set_secret`(key, value[, secret_store, kwargs])	Adds secret to the secret store (Only for superadmin users).

snorkelai.sdk.client.secrets.delete_secret

snorkelai.sdk.client.secrets.delete_secret(*key, secret_store='local_store', workspace_uid=1, kwargs=None*)

Deletes a secret from the secret store (Only for superadmin users).

Parameters

Name	Type	Default	Info
key	`str`		Key to reference the secret in the store.
secret_store	`str`	`'local_store'`	The secret store to delete the secret (only *local_store* supported now).
workspace_uid	`int`	`1`	The workspace uid for the secret.
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Return type

`None`

snorkelai.sdk.client.secrets.get_model_provider_status

snorkelai.sdk.client.secrets.get_model_provider_status(*model_provider*)

Gets the status of a model provider.

Parameters

Name	Type	Default	Info
model_provider	`ExternalLLMProvider`		The model provider to retrieve the status for.

Returns

The status of the model provider

Return type

`Dict[str, Any]`

snorkelai.sdk.client.secrets.list_integrations

snorkelai.sdk.client.secrets.list_integrations(*secret_store='local_store', workspace_uid=1, kwargs=None*)

Gets configuration status for each foundation model provider (Only for superadmin users).

Parameters

Name	Type	Default	Info
secret_store	`str`	`'local_store'`	The secret store to list the secret (only *local_store* supported now).
workspace_uid	`int`	`1`	The workspace uid for the secret.
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Return type

`None`

snorkelai.sdk.client.secrets.list_secrets

snorkelai.sdk.client.secrets.list_secrets(*secret_store='local_store', workspace_uid=1, kwargs=None*)

Gets all secret keys in a workspace (Only for superadmin users).

Parameters

Name	Type	Default	Info
secret_store	`str`	`'local_store'`	The secret store to list the secret (only *local_store* supported now).
workspace_uid	`int`	`1`	The workspace uid for the secret.
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Returns

All secret keys associated with a workspace

Return type

`List[str]`

snorkelai.sdk.client.secrets.set_secret

snorkelai.sdk.client.secrets.set_secret(*key, value, secret_store='local_store', kwargs=None*)

Adds secret to the secret store (Only for superadmin users).

Parameters

Name	Type	Default	Info
key	`str`		Key to reference the secret in the store.
value	`Union[str, Dict[str, str]]`		The secret being added.
secret_store	`str`	`'local_store'`	The secret store to add the secret (only *local_store* supported now).
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Return type

`None`

snorkelai.sdk.client.synthetic

Synthetic data related functions for generating synthetic data.

Functions

`augment_data`(data, model_name[, ...])	Augment each row of the data by the number of times specified and return a dataframe with the synthetic data as an additional column.
`augment_dataset`(dataset, x_uids, model_name)	Augment each row of the dataset by the number of times specified and return a dataframe containing only the synthetic data.

snorkelai.sdk.client.synthetic.augment_data

snorkelai.sdk.client.synthetic.augment_data(*data, model_name, runs_per_prompt=1, prompt='Rewrite the following text whilst retaining the core meaning.', sync=True, **fm_hyperparameters*)

Augment each row of the data by the number of times specified and return a dataframe with the synthetic data as an additional column.

Parameters

Name	Type	Default	Info
data	`Union[List[str], str]`		The data to augment.
model_name	`str`		The name of the foundation model to use.
runs_per_prompt	`int`	`1`	The number of times to augment each row.
prompt	`str`	`'Rewrite the following text whilst retaining the core meaning.'`	The prompt prefix to send to the foundation model together with each row.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
fm_hyperparameters	`Any`		Additional keyword arguments to pass

			to the foundation model such as temperature, max_tokens, etc.

Return type

`Union`[`DataFrame`, `str`]

Returns

- *df* – Dataframe containing the augmentations for the data points.
- *job_id* – The job id of the augment data job which can be used to monitor progress with sai.poll_job_status(job_id).

Examples

```
>>> sai.augment_data(["hello, how can I help you?", "sorry that is not
possible"], "openai/gpt-4")
   | text                           | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | hello, how can I help you?     | welcome, ask me a question to get
started   | 0.0113636364
1  | sorry that is not possible     | unfortunately you cannot do that
| 0.8901232123
```

```
>>> sai.augment_data(["hello, how can I help you?", "sorry that is not
possible"], "openai/gpt-4", runs_per_prompt=2)
   | text                             | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | hello, how can I help you?     | welcome, ask me a question to get
started   | 0.0113636364
1  | sorry that is not possible     | unfortunately you cannot do that
| 0.8901232123
0  | hello, how can I help you?     | Let me know how to get started.
| 0.2313232442
1  | sorry that is not possible     | bad luck, you cannot do that.
| 0.8313232442
```

snorkelai.sdk.client.synthetic.augment_dataset

snorkelai.sdk.client.synthetic.augment_dataset(*dataset, x_uids, model_name, runs_per_prompt=1, prompt='Your task is to rewrite the a set of text fields whilst retaining the core meaning. You should keep the same language and ensure each re-written field is of a similar length to the original.', fields=None, sync=True, **fm_hyperparameters*)

Augment each row of the dataset by the number of times specified and return a dataframe containing only the synthetic data. By default, all fields are augmented and the foundation model performs the augmentation of each row (all fields) in one inference step.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The name or UID of the dataset to generate a synthetic augmentation of.
x_uids	`List[str]`		The x_uids within the dataset to augment.
model_name	`str`		The name of the foundation model to use.
runs_per_prompt	`int`	`1`	The number of times to augment each row.
prompt	`str`	`'Your task is to`	The prompt passed to the

		`rewrite the a set of text fields whilst retaining the core meaning. You should keep the same language and ensure each re-written field is of a similar length to the original.'`	foundation model for each row. Note that by default, the prompt is appended with the fields to make the following: "Rewrite the following text fields whilst retaining the core meaning. You should keep the same language and ensure each re-written field is of a similar length to the original. Return your answer in a json format with the same keys as the fields: [field_1, field_2, ...] Here is the data you have to rewrite...". To override this default behavior, simply pass at least one field wrapped in parentheses, e.g. {field_1}, within the

			prompt and no additional text will be append to the prompt.
fields	`Optional[List[str]]`	`None`	The fields to augment. If not provided, all fields will be augmented.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
fm_hyperparameters	`Any`		Additional keyword arguments to pass to the foundation model such as temperature, max_tokens, etc.

Return type

`Union[DataFrame, str]`

Returns

- *df* – Dataframe containing the augmentations for the data points.
- *job_id* – The job id of the augment data job which can be used to monitor progress with sai.poll_job_status(job_id).

Examples

```
>>> sai.augment_dataset(dataset=1, x_uids=["0", "1"],
model_name="openai/gpt-4", runs_per_prompt=2)
   | subject                                | body
| perplexity
------------------------------------------------------------------------
-----------------------------------------------------------
0  | Fill in survey for $50 amazon voucher  | The email is asking you to
fill in a survey for an amazon voucher       | 0.891
1  | Hey it's Bob, free on Sat?             | The email is from your
friend Bob asking if you're free on Saturday     | 0.787
0  | Free survey for $50                    | Want a free $50 amazon
voucher? Fill in our survey.                     | 0.911
1  | No Plans on Sat, Bob?                  | Let's meet up on Sat. Bob.
| 0.991
```

snorkelai.sdk.client.users

User related functions.

Functions

`get_user`(user)	Get a user info by its username or user_uid.
`reset_password`(username, new_password)	Reset the password for the specified user to the specified new password.

snorkelai.sdk.client.users.get_user

snorkelai.sdk.client.users.get_user(*user*)

Get a user info by its username or user_uid.

Example

```
>>> sai.get_user("username")
{
    'username': 'username',
    'user_uid': 4,
    'default_view': 'standard',
    'role': 'standard',
    'is_active': True,
    'is_locked': False,
    'email': None,
    'timezone': None,
    'is_superadmin': False
}
```

Parameters

Name	Type	Default	Info
user	`Union[str, int]`		A valid Snorkel Flow user's username or user_uid.

Returns

The user info corresponding to the provided username/user_uid

Return type

`Dict[str, Any]`

snorkelai.sdk.client.users.reset_password

snorkelai.sdk.client.users.reset_password(*username, new_password*)

Reset the password for the specified user to the specified new password.

This functionality is only available to administrators as determined by the API key of the caller.

Parameters

Name	Type	Default	Info
username	`str`		Username of the user whose password you wish to reset.
new_password	`str`		The new password value you wish to set for this user.

Return type

`None`

snorkelai.sdk.client.utils

Utility functions.

Functions

`get_dataset_name`(dataset_uid)	Fetch the UID of a Dataset by name
`get_dataset_uid`(dataset)	Fetch the UID of a dataset by name or UID
`get_source_uid`(source_name)	Translate a source_name to a source_uid.
`get_workspace_name`(workspace_uid)	Fetch the name of a Workspace by UID
`get_workspace_uid`(workspace_name)	Fetch the UID of a Workspace by name
`poll_job_status`(job_id[, timeout, verbose])	Poll /jobs endpoint and print statuses.
`validate_traces`(trace_csv_file_path, ...)	Validate the traces in the trace CSV file.

snorkelai.sdk.client.utils.get_dataset_name

snorkelai.sdk.client.utils.get_dataset_name(*dataset_uid*)

Fetch the UID of a Dataset by name

Parameters

Name	Type	Default	Info
dataset_uid	`int`		UID of the dataset.

Returns

Name of the dataset

Return type

`str`

snorkelai.sdk.client.utils.get_dataset_uid

snorkelai.sdk.client.utils.get_dataset_uid(*dataset*)

Fetch the UID of a dataset by name or UID

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		Name or UID of the dataset.

Returns

UID of the dataset

Return type

`int`

Raises

ValueError – If a dataset doesn't exist.

snorkelai.sdk.client.utils.get_source_uid

snorkelai.sdk.client.utils.get_source_uid(*source_name*)

Translate a source_name to a source_uid.

Parameters

Name	Type	Default	Info
source_name	`str`		A valid source name.

Returns

The source_uid for source_name

Return type

`int`

snorkelai.sdk.client.utils.get_workspace_name

snorkelai.sdk.client.utils.get_workspace_name(*workspace_uid*)

Fetch the name of a Workspace by UID

Parameters

Name	Type	Default	Info
workspace_uid	`int`		UID of the workspace.

Returns

Name of the workspace

Return type

`str`

snorkelai.sdk.client.utils.get_workspace_uid

snorkelai.sdk.client.utils.get_workspace_uid(*workspace_name*)

Fetch the UID of a Workspace by name

Parameters

Name	Type	Default	Info
workspace_name	`str`		Name of the workspace.

Returns

UID of the workspace

Return type

`int`

snorkelai.sdk.client.utils.poll_job_status

snorkelai.sdk.client.utils.poll_job_status(*job_id, timeout=None, verbose=True*)

Poll /jobs endpoint and print statuses.

Parameters

Name	Type	Default	Info
job_id	`str`		UID of the job.
timeout	`Optional[timedelta]`	`None`	Optional polling timeout duration, jobs that exceed the timeout will be canceled.
verbose	`bool`	`True`	Optional argument to increase verbosity of the output logs.

Returns

Final job status response if job completes

Return type

`Dict[str, Any]`

Raises

- **JobFailedException** – If job fails
- **JobCancelledException** – If job is cancelled by user

Examples

```
>>> sai.poll_job_status(job_id)
{
    'uid': <job_id>,
    'job_type': <job_type>,
    'state': <state>, # completed or failed
    'enqueued_time': <timestamp>,
    'execution_start_time': <timestamp>,
    'end_time': <timestamp>,
    'application_uid':<application_uid>,
    'dataset_uid': <dataset_uid>,
    'node_uid': <node_uid>,
    'user_uid':<user_uid>,
    'workspace_uid': <workspace_uid>,
    'percent': <percent>, # Based on state
    'message':  <message>,
    'detail': <detail>,
    'pod_name':  <pod_name>,
    'function_name':  <function_name>,
    'process_id':  <process_id>,
    'timing': <timing_dict>
}
```

snorkelai.sdk.client.utils.validate_traces

snorkelai.sdk.client.utils.validate_traces(*trace_csv_file_path, trace_column*)

Validate the traces in the trace CSV file.

Parameters

Name	Type	Default	Info
trace_csv_file_path	`str`		The path to the trace CSV file.
trace_column	`str`		The column name containing the trace JSON data.

Returns

A dictionary containing the validation results.

Return type

`Dict[str, Any]`

snorkelai.sdk.develop

Object-oriented interfaces of Snorkel SDK

Classes

`AnnotationTask`(name, annotation_task_uid, ...)	Represents an annotation task within a Snorkel dataset for managing annotation workflows.
`Batch`(name, uid, dataset_uid, label_schemas, ...)	The Batch object represents an annotation batch in Snorkel Flow.
`Benchmark`(benchmark_uid, name, created_at, ...)	A benchmark is the collection of characteristics that you care about for a particular GenAI application, and the measurements you use to assess the performance against those characteristics.
`BenchmarkExecution`(benchmark_uid, ...)	Represents a single execution run of a benchmark for a dataset.
`Cluster`(cluster_uid, error_analysis_uid, name)	Provides methods for viewing and updating clusters and the ability to view datapoints assigned to a cluster.
`CodeEvaluator`(benchmark_uid, criteria_uid, ...)	An evaluator that uses custom Python code to assess an AI application's responses.
`Criteria`(benchmark_uid, criteria_uid, name, ...)	A criteria represents a specific characteristic or feature being evaluated as part of a benchmark.
`CsvExportConfig`([sep, quotechar, escapechar])	Benchmark execution CSV export configuration
`Dataset`(name, uid, mta_enabled)	The Dataset object represents a dataset in Snorkel Flow.
`ErrorAnalysis`(provenance, error_analysis_run_uid)	Provides methods for creating, monitoring, and retrieving results from error analysis clustering runs.
`Evaluator`(benchmark_uid, criteria_uid, ...)	Base class for all evaluators.
`JsonExportConfig`()	Benchmark execution JSON export configuration
`LabelSchema`(name, uid, dataset_uid, ...[, ...])	The LabelSchema object represents a label schema in Snorkel Flow.

`PromptEvaluator`(benchmark_uid, criteria_uid, ...)	An evaluator that uses LLM prompts to assess model outputs.
`Slice`(dataset, slice_uid, name[, ...])	Represents a slice within a Snorkel dataset for identifying and managing subsets of datapoints.
`User`(uid, username, role, default_view[, ...])	User management class for Snorkel SDK.
`UserRole`(value[, names, module, qualname, ...])	
`UserView`(value[, names, module, qualname, ...])	

snorkelai.sdk.develop.AnnotationTask

final class **snorkelai.sdk.develop.AnnotationTask**(*name, annotation_task_uid, dataset_uid, created_by_user_uid, created_at, num_required_annotations, description=None, label_schema_uids=None, x_uids=None, visible_columns=None*)

Bases: `Base`

Represents an annotation task within a Snorkel dataset for managing annotation workflows.

An annotation task defines a set of datapoints that need to be annotated, along with the annotation form, user assignments, and task configuration. This class provides methods for creating, retrieving, updating, and managing annotation tasks.

AnnotationTask objects should not be instantiated directly - use the `create()` or `get()` class methods instead.

__init__

__init__(*name, annotation_task_uid, dataset_uid, created_by_user_uid, created_at, num_required_annotations, description=None, label_schema_uids=None, x_uids=None, visible_columns=None*)

Create an AnnotationTask object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods.

Parameters

Name	Type	Default	Info
name	`str`		The name of the annotatio task.
annotation_task_uid	`int`		The unique identifier for t annotation ta
dataset_uid	`int`		The UID of th dataset that t

			annotation ta belongs to.
created_by_user_uid	`int`		The UID of th user who created the annotation ta
created_at	`datetime`		The timestam when the annotation ta was created.
num_required_annotations	`int`		The number c required annotator an annotation fo this annotatic task.
description	`Optional[str]`	`None`	A description the annotatio task, by defau None.
label_schema_uids	`Optional[List[int]]`	`None`	Change annotation form to the n list of label schema uids. will remove annotations c label schema uids that are not in the nev form.
x_uids	`Optional[List[str]]`	`None`	List of datapc UIDs in this

			annotation ta by default None.
visible_columns	`Optional[List[str]]`	`None`	List of columr names to sho in get_datafram If None, all columns are shown.

Methods

`__init__`(name, annotation_task_uid, ...[, ...])	Create an AnnotationTask object in-memory with necessary properties.
`add_annotations`(annotations[, user_format])	Add annotations to an annotation task.
`add_assignees`(users, x_uids)	Assign all of the listed users to the listed datapoints in the annotation task.
`add_datapoints`(x_uids)	Add datapoints to the annotation task.
`add_label_schemas`(label_schema_uids)	Add label schemas to the annotation task.
`commit_annotations`([x_uids])	Commit annotations found on the specified x_uids to ground truth using majority vote.
`create`(dataset_uid, name[, description])	Create an annotation task.
`delete`(annotation_task_uid)	Delete an annotation task.
`get`(annotation_task_uid)	Get an annotation task by UID.
`get_annotation_status`([user_format])	Fetch the task columns (assignees, status) for all the datapoints in an annotation task.
`get_annotations`([user_format, user_uids, ...])	Get annotations from an annotation task, filtered by the uids specified.

`get_annotator_stats`([user, event_name])	Get aggregated annotator statistics for this annotation task.
`get_dataframe`([limit, offset])	Fetch the dataset columns for all the datapoints in an annotation task.
`list`(dataset)	List all annotation tasks for a given dataset.
`list_user_assignments`(users[, user_format])	Get user assignments in an annotation task.
`mark_datapoints_complete`([x_uids])	Mark datapoints that have ground truth completed for all required label schemas.
`remove_assignees`([users, x_uids])	Remove all of the listed users from the listed datapoints in the annotation task.
`remove_datapoints`(x_uids)	Remove datapoints from the annotation task.
`update`([name, description, ...])	Update an annotation task.

Attributes

`annotation_form`	Return the annotation form of the annotation task
`created_at`	Return the creation timestamp of the annotation task
`created_by_user_uid`	Return the UID of the user who created the annotation task
`dataset_uid`	Return the UID of the dataset that the annotation task belongs to
`description`	Return the description of the annotation task
`label_schema_uids`	Return the list of label schema UIDs associated with this annotation task.
`name`	Return the name of the annotation task
`num_required_annotations`	Return the number of required annotator and annotation for this annotation task
`uid`	Return the UID of the annotation task

`visible_columns`	Return the list of visible columns for get_dataframe().
`x_uids`	Return the list of datapoint UIDs in this annotation task

add_annotations

add_annotations(*annotations, user_format=True*)

Add annotations to an annotation task.

Parameters

Name	Type	Default	Info
annotations	`Union[DataFrame, List[Dict]]`		Annotations to add to the task. Can be provided in one of two formats: 1. **DataFrame**: A pandas DataFrame with annotation data 2. **List of Dicts**: A list of dictionaries with annotation parameters
user_format	`bool`	`True`	True if annotation labels in data are in user format, otherwise they must be raw label values.

Return type

`None`

Examples

DataFrame Input:

```python
import pandas as pd
df = pd.DataFrame([
    {
        'x_uid': 'doc::1',
        'dataset_uid': 1001,
        'label_schema_uid': 101,
        'label': 'positive',
        'metadata': {'confidence': 0.95},
        'freeform_text': None
    },
    {
        'x_uid': 'doc::2',
        'dataset_uid': 1001,
        'label_schema_uid': 101,
        'label': 'negative',
        'metadata': {'confidence': 0.87},
        'freeform_text': None
    }
])
annotation_task.add_annotations(df)
```

List of Dictionaries Input:

```python
annotations_list = [
    {
        'x_uid': 'doc::3',
        'dataset_uid': 1002,
        'label_schema_uid': 102,
        'label': {'spans': [[0, 10, 'PERSON'], [15, 25, 'ORG']]},
        'metadata': {'annotator': 'user_123'},
        'freeform_text': None
    },
    {
        'x_uid': 'doc::4',
        'dataset_uid': 1002,
        'label_schema_uid': 103,
        'label': {},  # Empty for text annotations
        'metadata': {},
        'freeform_text': 'This document discusses climate change
impacts.'
    }
]
annotation_task.add_annotations(annotations_list)
```

Dictionaries of different label types:

```
# Single-label classification
single_label = [{'label': 'category_a', 'x_uid': 'doc::6',
'dataset_uid': 1003, 'label_schema_uid': 101}]

# Multi-label classification
multi_label = [{'label': ['tag1', 'tag2', 'tag3'], 'x_uid':
'doc::7', 'dataset_uid': 1003, 'label_schema_uid': 102}]

# Sequence tagging (NER)
sequence_tags = [{'label': [[0, 5, 'B-PER'], [6, 15, 'B-LOC']],
'x_uid': 'doc::8', 'dataset_uid': 1003, 'label_schema_uid': 103}]

# Text annotation (freeform)
text_annotation = [{'label': {}, 'freeform_text': 'User feedback
here', 'x_uid': 'doc::9', 'dataset_uid': 1003, 'label_schema_uid':
104}]

annotation_task.add_annotations(single_label)
annotation_task.add_annotations(multi_label)
annotation_task.add_annotations(sequence_tags)
annotation_task.add_annotations(text_annotation)
```

Notes

- All annotations must belong to label schemas associated with this annotation task
- The *x_uid* must correspond to datapoints in the task's dataset
- For text-based labels (*is_text_label=True*), use *freeform_text* instead of *label*
- For structured labels, use the *label* field with appropriate format for the label type
- Metadata is optional and can contain arbitrary key-value pairs
- Timestamps (*ts*) are auto-generated if not provided

Raises

- **ValueError** – If annotation format is invalid or contains missing required fields
- **UserInputError** – If x_uid is empty or label_schema_uid is not associated with this task

add_assignees

add_assignees(*users*, *x_uids*)

Assign all of the listed users to the listed datapoints in the annotation task.

Parameters

Name	Type	Default	Info
users	`List[Union[int, str]]`		List of users to assign to the datapoints. Can be user UIDs (int) or usernames (str).
x_uids	`List[str]`		List of datapoint UIDs to assign the users to.

Raises

ValueError – If users or x_uids are empty, or if user input is invalid

Return type

`None`

Examples

Add assignees using user UIDs:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.add_assignees(users=[101, 102, 103], x_uids=["doc::1",
"doc::2", "doc::3"])
```

Add assignees using usernames:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.add_assignees(users=["alice", "bob", "charlie"], x_uids=
["doc::1", "doc::2", "doc::3"])
```

Add assignees using mixed usernames and UIDs:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.add_assignees(users=["alice", 102, "charlie"], x_uids=
["doc::1", "doc::2", "doc::3"])
```

add_datapoints

add_datapoints(*x_uids*)

Add datapoints to the annotation task.

Parameters

Name	Type	Default	Info
x_uids	`List[str]`		List of datapoint UIDs to add to the annotation task.

Return type

`None`

add_label_schemas

add_label_schemas(*label_schema_uids*)

Add label schemas to the annotation task.

Label schemas will be displayed in the order in which they are added.

Parameters

Name	Type	Default	Info
label_schema_uids	`List[int]`		List of label schema UIDs to add to the annotation task.

Raises

ValueError – If label_schema_uids is empty, label schemas are not existing in the dataset or if updating the annotation task fails

Return type

`None`

Example

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.add_label_schemas(label_schema_uids=[1, 2, 3])
```

commit_annotations

commit_annotations(*x_uids=None*)

Commit annotations found on the specified x_uids to ground truth using majority vote.

If no x_uids are provided, commit all that are ready for review. This also updates the datapoint status from ready for review to complete. Note: if x_uids are not in ready for review state, they will be skipped.

Parameters

Name	Type	Default	Info
x_uids	`Optional[List[str]]`	`None`	The datapoints to commit annotations from. If None, all x_uids that are ready for review will have their annotations committed. By default None.

Returns

A dictionary with two keys: - “completed”: List of x_uids whose annotations were committed to GT and status marked complete - “skipped”: List of x_uids that were skipped (not in ready for review state)

Return type

`Dict[str, List[str]]`

create

classmethod **create**(*dataset_uid, name, description=None*)

Create an annotation task.

> NOTE
>
> This method only accepts *dataset_uid*, *name*, and *description* as parameters. Other properties (such as label schema UIDs, datapoint UIDs, and assignees) can be set later through other methods.

Parameters

Name	Type	Default	Info
dataset_uid	`int`		The UID of the dataset for the annotation task.
name	`str`		The name of the annotation task.
description	`Optional[str]`	`None`	A description of the annotation task, by default None.

Returns

The created annotation task object

Return type

AnnotationTask

delete

classmethod **delete**(*annotation_task_uid*)

Delete an annotation task.

Parameters

Name	Type	Default	Info
annotation_task_uid	`int`		The UID of the annotation task to delete.

Return type

`None`

get

classmethod **get**(*annotation_task_uid*)

Get an annotation task by UID.

Parameters

Name	Type	Default	Info
annotation_task_uid	`int`		The UID of the annotation task to retrieve.

Returns

The annotation task object

Return type

AnnotationTask

get_annotation_status

get_annotation_status(*user_format=True*)

Fetch the task columns (assignees, status) for all the datapoints in an annotation task.

Parameters

Name	Type	Default	Info
user_format	`bool`	`True`	If True, assignee names are returned instead of uids.

Returns

A DataFrame with columns: x_uid (this is the index), assignees (list of user UIDs or usernames), status (str) The DataFrame will have one row per datapoint in the annotation task

Example:

Data Point ID Assignee(s) Status ———- ———- ———- doc::1 [101, 102] IN_ANNOTATION doc::2 [103] READY_FOR_REVIEW doc::3 [101, 104] COMPLETED doc::4 [] NEEDS_ASSIGNEES

Return type

`pd.DataFrame`

get_annotations

get_annotations(*user_format=True, user_uids=None, label_schema_uids=None, source_uids=None*)

Get annotations from an annotation task, filtered by the uids specified.

Parameters

Name	Type	Default	Info
user_format	`bool`	`True`	If True, convert raw label value to label names.
user_uids	`Optional[List[int]]`	`None`	List of user UIDs to filter annotations

			by, by default None.
label_schema_uids	`Optional[List[int]]`	`None`	List of label schema UIDs to filter annotations by, by default None.
source_uids	`Optional[List[int]]`	`None`	List of source UIDs to filter annotations by, by default None.

Returns

DataFrame containing the filtered annotations with label values transformed to label names if user_format is True

Return type

`pd.DataFrame`

get_annotator_stats

get_annotator_stats(*user=None, event_name=None*)

Get aggregated annotator statistics for this annotation task.

This method returns annotation time statistics per user, per datapoint, aggregated from activity events.

Parameters

Name	Type	Default	Info
user	`Optional[int]`	`None`	Filter by specific user UID. If None, returns stats for all users who have activity events recorded for this task.
event_name	`Optional[str]`	`None`	Filter by event name. Valid values are: - “ANNOTATION_DURATION”:

			Time spent submitting labels for a datapoint If None, returns stats for all event types.

Returns

DataFrame where each row represents a unique (user, datapoint, event_type) combination. Returns empty DataFrame if no matching activity events exist.

Columns: Multi-index levels: - __DATAPOINT_UID: Datapoint identifier - user_uid: User who performed the annotation - event_name: Type of event (ANNOTATION_DURATION)

Columns: - total_time_seconds: Active time (excludes paused periods)

Return type

`pd.DataFrame`

Notes

- Only includes users who have recorded activity events; task assignment alone is not sufficient to appear in results.
- If a specified user has no events, returns an empty DataFrame (no error).
- The multi-index allows easy joining with other DataFrames on (__DATAPOINT_UID, user_uid, event_name).

Examples

Get all annotator stats for a task:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
stats_df = task.get_annotator_stats()
```

Get stats for a specific user:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
stats_df = task.get_annotator_stats(user=101)
```

Get stats for a specific event type:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
stats_df =
task.get_annotator_stats(event_name="ANNOTATION_DURATION")
```

get_dataframe

get_dataframe(*limit=None, offset=0*)

Fetch the dataset columns for all the datapoints in an annotation task.

Parameters

Name	Type	Default	Info
limit	`Optional[int]`	`None`	The max number of rows to return. If None, all rows will be returned.
offset	`int`	`0`	Rows will be returned starting at this index.

Returns

DataFrame containing the dataset data

Return type

`pd.DataFrame`

list

classmethod list(*dataset*)

List all annotation tasks for a given dataset.

Parameters

Name	Type	Default	Info

dataset	`Union[str, int]`		The dataset UID or dataset object to list annotation tasks for.

Returns

A list of annotation task objects

Return type

List[AnnotationTask]

list_user_assignments

list_user_assignments(*users, user_format=True*)

Get user assignments in an annotation task.

Parameters

Name	Type	Default	Info
users	`List[Union[int, str]]`		List of users to fetch annotation assignments for. Can be user UIDs (int) or usernames (str).
user_format	`bool`	`True`	If true, return user names as keys; if false, return user UIDs as keys.

Returns

A dictionary with user keys (names if user_format is True, UIDs otherwise) and values containing lists of datapoint_uids that the user is assigned to

Example:

```
assignments = {
    "Dr Bubbles": ["doc::1", "doc::2"],
    "Rebekah": ["doc::5"],
    "Hiromu": [],
}
```

Return type

`Dict[str | int, List[str]]`

Raises

ValueError – If user_uids is empty or if fetching assignments fails

Examples

Get assignments using user UIDs, returning usernames as keys:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
assignments = task.list_user_assignments(users=[101, 102, 103])
# Returns dictionary with usernames as keys
# {'alice': ['doc::1', 'doc::2'], 'bob': ['doc::3'], 'charlie': []}
```

Get assignments using usernames, returning usernames as keys:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
assignments = task.list_user_assignments(users=['alice', 'bob',
'charlie'])
# Returns dictionary with usernames as keys
# {'alice': ['doc::1', 'doc::2'], 'bob': ['doc::3'], 'charlie': []}
```

Get assignments using user UIDs, returning UIDs as keys:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
assignments = task.list_user_assignments(users=[101, 102],
user_format=False)
# Returns dictionary with user UIDs as keys
# {101: ['doc::1', 'doc::2'], 102: ['doc::3']}
```

Get assignments using mixed input (usernames and UIDs):

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
assignments = task.list_user_assignments(users=['alice', 102,
'charlie'])
```

mark_datapoints_complete

mark_datapoints_complete(*x_uids=None*)

Mark datapoints that have ground truth completed for all required label schemas.

This method searches for datapoints that have ground truth present for all required label schemas and automatically advances their status to COMPLETED.

Parameters

Name	Type	Default	Info
x_uids	`Optional[List[str]]`	`None`	Optional list of datapoint UIDs to check. If None, checks all datapoints in the annotation task that are not already COMPLETED.

Returns

A dictionary with two keys: - “completed”: List of x_uids that were successfully marked completed - “skipped”: List of x_uids that were skipped (missing GT for some label schemas)

Return type

`Dict[str, List[str]]`

Examples

Complete all eligible datapoints in the task:

```python
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
result = task.mark_datapoints_complete()
print(f"Completed: {len(result['completed'])}, Skipped:
{len(result['skipped'])}")
```

Complete specific datapoints:

```python
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
result = task.mark_datapoints_complete(x_uids=["doc::1", "doc::2",
"doc::3"])
print(f"Completed: {result['completed']}")
print(f"Skipped: {result['skipped']}")
```

Notes

- Only datapoints with ground truth for ALL required label schemas will be marked completed
- Datapoints already in COMPLETED status are skipped
- The annotator_status remains unchanged

remove_assignees

remove_assignees(*users=None, x_uids=None*)

Remove all of the listed users from the listed datapoints in the annotation task.

If both users and x_uids are None, it will remove all assignees from all datapoints in the task. This is a non-destructive operation – it removes the assignments but retains the annotations.

Parameters

Name	Type	Default	Info
users	`Optional[List[Union[int, str]]]`	`None`	A list of users to remove from listed datapoints, by default None. Can be user UIDs (int) or usernames (str). If None, all users assigned to

			listed datapoints will be removed from those datapoints.
x_uids	`Optional[List[str]]`	`None`	A list of the x_uids of datapoints to remove users from, by default None. If None, listed users will be removed from all the datapoints they are assigned to.

Raises

ValueError – If fetching or deleting assignments fails

Return type

`None`

Examples

Remove specific users from specific datapoints using UIDs:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.remove_assignees(users=[101, 102], x_uids=["doc::1",
"doc::2"])
```

Remove specific users from specific datapoints using usernames:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.remove_assignees(users=["alice", "bob"], x_uids=["doc::1",
"doc::2"])
```

Remove specific users from specific datapoints using mixed identifiers:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.remove_assignees(users=["alice", 102], x_uids=["doc::1",
"doc::2"])
```

Remove specific users from all datapoints they are assigned to:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.remove_assignees(users=["alice", "bob"])
```

Remove all users from specific datapoints:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.remove_assignees(x_uids=["doc::1", "doc::2"])
```

Remove all assignees from all datapoints in the task:

```
from snorkelai.sdk.develop import AnnotationTask
task = AnnotationTask.get(annotation_task_uid=123)
task.remove_assignees()
```

remove_datapoints

remove_datapoints(*x_uids*)

Remove datapoints from the annotation task.

Parameters

Name	Type	Default	Info
x_uids	`List[str]`		List of datapoint UIDs to remove from the annotation task.

Return type

`None`

update

update(*name=None, description=None, label_schema_uids=None, num_required_annotations=None, visible_columns=None*)

Update an annotation task.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new nam for the annotation ta by default None.
description	`Optional[str]`	`None`	The new description fc the annotatio task, by defau None.
label_schema_uids	`Optional[List[int]]`	`None`	Change annotation form to the n list of label schema uids. will remove annotations c label schema uids that are not in the nev form.
num_required_annotations	`Optional[int]`	`None`	The number c required annotator an

			annotation fo each datapoi
visible_columns	`Optional[List[str]]`	`None`	List of columr names to sho in get_datafram If None, all columns are shown.

Return type

`None`

property **annotation_form**: *AnnotationForm*

Return the annotation form of the annotation task

property **created_at**: *datetime*

Return the creation timestamp of the annotation task

property **created_by_user_uid**: *int*

Return the UID of the user who created the annotation task

property **dataset_uid**: *int*

Return the UID of the dataset that the annotation task belongs to

property **description**: *str | None*

Return the description of the annotation task

property **label_schema_uids**: *List[int]*

Return the list of label schema UIDs associated with this annotation task.

property **name**: *str*

Return the name of the annotation task

property **num_required_annotations**: *int*

Return the number of required annotator and annotation for this annotation task

property **uid**: *int*

Return the UID of the annotation task

property **visible_columns**: *List[str] | None*

Return the list of visible columns for get_dataframe().

If None, all columns are shown. If set, only these columns will be returned by get_dataframe().

property x_uids*: List[str]*

Return the list of datapoint UIDs in this annotation task

snorkelai.sdk.develop.Batch

final class **snorkelai.sdk.develop.Batch**(*name, uid, dataset_uid, label_schemas, batch_size, ts, x_uids*)

Bases: `Base`

The Batch object represents an annotation batch in Snorkel Flow. Currently, this interface only represents Dataset-level (not Node-level) annotation batches.

__init__

__init__(*name, uid, dataset_uid, label_schemas, batch_size, ts, x_uids*)

Create a batch object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods.

Parameters

Name	Type	Default	Info
name	`str`		The name of the batch.
uid	`int`		The UID for the batch within Snorkel Flow.
dataset_uid	`int`		The UID for the dataset within Snorkel Flow.
label_schemas	`List[LabelSchema]`		The list of label schemas associated with this batch.
batch_size	`int`		The number of examples in the batch.
ts	`datetime`		The timestamp at which the batch was created.
x_uids	`List[str]`		The UIDs for the examples in the batch.

Methods

`__init__`(name, uid, dataset_uid, ...)	Create a batch object in-memory with necessary properties.
`commit`(source_uid[, label_schema_uids])	Commit a source on a batch as ground truth.
`create`(dataset_uid[, name, assignees, ...])	Create one or more annotation batches for a dataset.
`delete`(batch_uid)	Delete an annotation batch by its UID.
`export`(path[, selected_fields, ...])	Export the batch to a zipped CSV file.
`get`(batch_uid)	Retrieve an annotation batch by its UID.
`get_dataframe`([selected_fields, ...])	Get a pandas DataFrame representation of the batch.
`update`([name, assignees, expert_source_uid])	Update properties of the annotation batch.

Attributes

`batch_size`	The number of examples in the batch.
`dataset_uid`	The UID for the dataset within Snorkel Flow.
`label_schemas`	The list of label schemas associated with this batch.
`name`	The name of the batch.
`ts`	The timestamp at which the batch was created.
`uid`	The UID for the batch within Snorkel Flow.
`x_uids`	The UIDs for the examples in the batch.

commit

commit(*source_uid, label_schema_uids=None*)

Commit a source on a batch as ground truth.

Parameters

Name	Type	Default	Info
source_uid	`int`		The UID for the source on the batch.
label_schema_uids	`Optional[List[int]]`	`None`	The label schema UIDs to commit, defaults to all label schemas if not set.

Return type

`None`

create

classmethod **create**(*dataset_uid, name=None, assignees=None, label_schemas=None, batch_size=None, randomize=False, random_seed=123, selection_strategy=None, split=None, x_uids=None, filter_by_x_uids_not_in_batch=False, divide_x_uids_evenly_to_assignees=False*)

Create one or more annotation batches for a dataset.

Typically, Dataset.create_batches() is the recommended entrypoint for creating batches.

Parameters

Name	Type	Defaul
dataset_uid	`int`	
name	`Optional[str]`	`None`

assignees	`Optional[List[int]]`	`None`
label_schemas	`Optional[List[LabelSchema]]`	`None`
batch_size	`Optional[int]`	`None`
randomize	`Optional[bool]`	`False`
random_seed	`Optional[int]`	`123`
selection_strategy	`Optional[SelectionStrategy]`	`None`
split	`Optional[str]`	`None`
x_uids	`Optional[List[str]]`	`None`
filter_by_x_uids_not_in_batch	`Optional[bool]`	`False`
divide_x_uids_evenly_to_assignees	`Optional[bool]`	`False`

Returns

The list of created batches

Return type

List[Batch]

delete

classmethod **delete**(batch_uid)

Delete an annotation batch by its UID.

Parameters

Name	Type	Default	Info
batch_uid	`int`		The UID for the batch within Snorkel Flow.

Return type

`None`

export

export(path, selected_fields=None, include_annotations=False, include_ground_truth=False, max_rows=10000, csv_delimiter=',', quote_char='"', escape_char='\\\\')

Export the batch to a zipped CSV file.

Parameters

Name	Type	Default	Info

path	`Union[str, Path]`		The path to the zipped CSV file. If the path does not end in .zip, it will be appended to the path.
selected_fields	`Optional[List[str]]`	`None`	A list of fields to export. If not set, all fields will be exported.
include_annotations	`bool`	`False`	Whether to include annotations in the export.
include_ground_truth	`bool`	`False`	Whether to include ground truth in the export.
max_rows	`int`	`10000`	The maximum number of rows to export.
csv_delimiter	`str`	`','`	The delimiter to use for CSV fields.
quote_char	`str`	`'"'`	The character to use for quoted fields in the CSV.
escape_char	`str`	`'\\\\'`	The character to use for

			escaping special characters in the CSV.

Returns

The path to the zipped CSV file

Return type

`pathlib.Path`

get

classmethod **get**(*batch_uid*)

Retrieve an annotation batch by its UID.

Parameters

Name	Type	Default	Info
batch_uid	`int`		The UID for the batch within Snorkel Flow.

Returns

The batch object

Return type

Batch

get_dataframe

get_dataframe(*selected_fields=None, include_annotations=False, include_ground_truth=False, max_rows=10000*)

Get a pandas DataFrame representation of the batch.

Parameters

Name	Type	Default	Info
selected_fields	`Optional[List[str]]`	`None`	A list of fields to include in the DataFrame. If not set, all fields will be included.
include_annotations	`bool`	`False`	Whether to include annotations in the DataFrame.
include_ground_truth	`bool`	`False`	Whether to include ground truth in the DataFrame.
max_rows	`int`	`10000`	The maximum number of rows to include in the DataFrame.

Returns

The pandas DataFrame representation of the batch

Return type

`pd.DataFrame`

update

update(*name=None, assignees=None, expert_source_uid=None*)

Update properties of the annotation batch.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new name of the batch.
assignees	`Optional[List[int]]`	`None`	The user UIDs for the new assignees of the batches.
expert_source_uid	`Optional[int]`	`None`	The UID for the new expert source of the batches.

Return type

`None`

property **batch_size**: *int*

The number of examples in the batch.

property **dataset_uid**: *int*

The UID for the dataset within Snorkel Flow.

property **label_schemas**: *List[LabelSchema]*

The list of label schemas associated with this batch.

property **name**: *str*

The name of the batch.

property **ts**: *datetime*

The timestamp at which the batch was created.

property **uid**: *int*

The UID for the batch within Snorkel Flow.

property **x_uids**: *List[str]*

The UIDs for the examples in the batch.

snorkelai.sdk.develop.Benchmark

final class snorkelai.sdk.develop.Benchmark(*benchmark_uid, name, created_at, updated_at, archived, description=None*)

Bases: `Base`

A benchmark is the collection of characteristics that you care about for a particular GenAI application, and the measurements you use to assess the performance against those characteristics. It consists of the following elements:

- **Reference prompts**: A set of prompts used to evaluate the model's responses.
- **Slices:** Subsets of reference prompts focusing on specific topics.
- **Criteria**: Key characteristics that represent the features being optimized for evaluation.
- **Evaluators:** Functions that assess whether a model's output satisfies the criteria.

Read more in the Evaluation overview.

Using the `Benchmark` class requires the following import:

```
from snorkelai.sdk.develop import Benchmark
```

__init__

__init__(*benchmark_uid, name, created_at, updated_at, archived, description=None*)

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark from which you want to get data. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/`

			indicates a benchmark with `benchmark_uid` of `100`.
name	`str`		The name of the benchmark.
description	`Optional[str]`	`None`	The description of the benchmark.
created_at	`datetime`		The timestamp when the benchmark was created.
updated_at	`datetime`		The timestamp when the benchmark was last updated.
archived	`bool`		Whether the benchmark is archived.

Methods

`__init__`(benchmark_uid, name, created_at, ...)	
`archive`()	Archives the benchmark, hiding it from the UI and SDK list method.
`create`(name, dataset_uid[, description])	Creates a new benchmark.
`delete`(benchmark_uid)	Deletion of a benchmark is not implemented.
`execute`([splits, criteria_uids, name])	Executes the benchmark against the associated dataset.
`export_config`(filepath[, format])	Exports a benchmark configuration to the specified format and writes to the provided filepath.
`export_latest_execution`(filepath[, config])	Export the latest benchmark execution with all its associated data.
`get`(benchmark_uid)	Gets a benchmark by its unique identifier.
`list`(workspace_uid[, include_archived])	Lists all benchmarks for a given workspace.

`list_criteria`([include_archived])	Retrieves all criteria for this benchmark.
`list_executions`([include_archived])	Retrieves all benchmark executions for this benchmark.
`update`([name, description, archived])	Updates the benchmark with the given parameters.

Attributes

`archived`	Return whether the benchmark is archived
`benchmark_uid`	Return the UID of the benchmark
`created_at`	Return the timestamp when the benchmark was created
`description`	Return the description of the benchmark
`name`	Return the name of the benchmark
`uid`	Return the UID of the benchmark
`updated_at`	Return the timestamp when the benchmark was last updated

archive

archive()

Archives the benchmark, hiding it from the UI and SDK list method.

Use `snorkelai.sdk.develop.benchmarks.Benchmark.list()` with `include_archived=True` to view archived benchmarks.

Return type

`None`

create

static **create**(*name, dataset_uid, description=None*)

Creates a new benchmark. The created benchmark does not include any default criteria or evaluators.

Parameters

Name	Type	Default	Info
name	`str`		The name of the benchmark.
dataset_uid	`int`		The unique identifier of the dataset t the benchmark. The `dataset_uid` c the **`snorkelai.sdk.develop.datase`** method.
description	`Optional[str]`	`None`	The description of the benchmark.

Returns

A Benchmark object representing the created benchmark.

Return type

Benchmark

delete

classmethod **delete**(*benchmark_uid*)

Deletion of a benchmark is not implemented.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark.

Return type

`None`

execute

execute(*splits=None, criteria_uids=None, name=None*)

Executes the benchmark against the associated dataset. For each criteria, evaluation scores are computed for each datapoint and aggregate metrics are computed across all datapoints.

Parameters

Name	Type	Default	Info
splits	`Optional[List[str]]`	`None`	The splits to execute the benchmark on. If not provided, will default to [“train”, “valid”].
criteria_uids	`Optional[List[int]]`	`None`	The criteria to execute the benchmark on. If not provided, will default to all criteria for the benchmark.
name	`Optional[str]`	`None`	The name of the execution. If not provided, will default to “Run <number>” based on the number of previous executions.

Returns

The execution object.

Return type

BenchmarkExecution

export_config

export_config(*filepath, format=BenchmarkExportFormat.JSON*)

Exports a benchmark configuration to the specified format and writes to the provided filepath.

This method exports the complete benchmark configuration, including all criteria, evaluators, and metadata. The exported configuration can be used for:

- Version control of benchmark definitions.
- Sharing benchmarks across teams.
- Integration with CI/CD pipelines.
- Backing up evaluation configurations.

Parameters

Name	Type	Default	
filepath	`str`		C p e d d w c it e
format	`BenchmarkExportFormat`	`<BenchmarkExportFormat.JSON: 'json'>`	T f e c C o is s

Raises

- **NotImplementedError** – If an unsupported export format is specified.
- **ValueError** – If the benchmark_uid is None or invalid.

Return type

`None`

Example

Example 1

Export a benchmark configuration to JSON:

```
benchmark = Benchmark.get(100)
benchmark.export_config("benchmark_config.json")
```

Example 1 output

The exported JSON file contains:

```
{
  "criteria": [
    {
      "criteria_uid": 101,
      "benchmark_uid": 100,
      "name": "Example Readability",
      "description": "Evaluates how easy the response is to read
and understand.",
      "state": "ACTIVE",
      "output_format": {
        "metric_label_schema_uid": 200,
        "rationale_label_schema_uid": 201
      },
      "metadata": {
        "version": "1.0"
      },
      "created_at": "2025-04-01T14:30:00.123456Z",
      "updated_at": "2025-04-01T14:35:10.654321Z"
    }
  ],
  "evaluators": [
    {
      "evaluator_uid": 301,
      "name": "Readability Evaluator (LLM)",
      "description": "Uses an LLM prompt to assess readability.",
      "criteria_uid": 101,
      "type": "Prompt",
      "prompt_workflow_uid": 401,
      "parameters": null,
      "metadata": {
        "default_prompt_config": {
          "name": "Readability Prompt v1",
          "model_name": "google/gemini-1.5-pro-latest",
          "system_prompt": "You are an expert evaluator assessing
text readability.",
          "user_prompt": "..."
        }
      },
      "created_at": "2025-04-01T15:00:00.987654Z",
      "updated_at": "2025-04-01T15:05:00.123123Z"
    }
  ],
  "metadata": {
    "name": "Sample Benchmark Set",
    "description": "A benchmark set including example
evaluations.",
    "created_at": "2025-04-01T14:00:00.000000Z",
    "created_by": "user@example.com"
    }
                }
```

After exporting your benchmark, you can use it to evaluate data from your GenAI application iteratively, allowing you to measure and refine your LLM system.

export_latest_execution

export_latest_execution(*filepath, config=None*)

Export the latest benchmark execution with all its associated data.

This method exports the most recent benchmark execution, including all evaluation results and metadata. The exported dataset contains:

- Benchmark metadata for the associated benchmark
- Execution metadata for this execution
- Each datapoint lists its evaluation score, which includes:
  - The evaluator outputs
  - Rationale
  - Agreement with ground truth
- Each datapoint lists its slice membership(s)
- (CSV exports only) Uploaded user columns and ground truth

The export includes all datapoints without filtering or sampling. Some datapoints may have missing evaluation scores if the benchmark was not executed against them (for example, datapoints in the test split).

Parameters

Name	Type	Default	Info
filepath	`str`		Output file path for exported data.
config	`Union[JsonExportConfig, CsvExportConfig, None]`	`None`	A `JsonExportConfig` or `CsvExportConfig` object. If not provided, JSON will be used by

			default. No additional configuration is required for JSON exports. For CSV exports, the following parameters are supported: • `sep`: The separator between columns. Default: `,`. • `quotechar`: The character used to quote fields. Default: `"`. • `escapechar`: The character used to escape special characters. Default: `\`.

Return type

`None`

Example

Example 1

Export the latest benchmark execution to JSON:

```
benchmark = Benchmark.get(100)
benchmark.export_latest_execution("benchmark_execution.json")
```

Example 1 return

The exported JSON file contains:

```
{
    "benchmark_metadata": {
        "uid": 100,
        "name": "Example Benchmark",
        "description": "A benchmark for testing model performance",
        "created_at": "2025-01-01T12:00:00Z",
        "created_by": "user@example.com"
    },
    "execution_metadata": {
        "uid": 1,
        "name": "Latest Run",
        "created_at": "2025-01-01T12:00:00Z",
        "created_by": "user@example.com"
    },
    "data": [
        {
            "x_uid": "doc::0",
            "scores": [
               {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "RATIONALE",
                    "value": "The response is clear and well-
structured",
                    "error": ""
               },
               {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "EVAL",
                    "value": 0.85,
                },
                {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "AGREEMENT",
                    "value": 1.0
                }
            ],
            "slice_membership": ["test_set"]
        },
        {
            "x_uid": "doc::1",
            "scores": [
                {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "EVAL",
                    "value": 0.92,
                }
            ],
            "slice_membership": ["test_set"]
        }
    ],
    "slices": [
        {
            "id": "None",
            "display_name": "All Datapoints",
            "reserved_slice_type": "global"
        },
        {
            "id": "-1",
            "display_name": "No Slice",
            "reserved_slice_type": "no_slice"
        },
        {
            "id": "5",
            "display_name": "Your Slice",
            "reserved_slice_type": "regular_slice"
        }
    ]
}
```

get

static **get**(*benchmark_uid*)

Gets a benchmark by its unique identifier.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark from which you want to get data. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.

Returns

A Benchmark object representing the benchmark with the given `benchmark_uid`.

Return type

Benchmark

list

static **list**(*workspace_uid, include_archived=False*)

Lists all benchmarks for a given workspace.

Parameters

Name	Type	Default	Info
workspace_uid	`int`		The unique identifier of the workspace fron list benchmarks. The `workspace_uid` can the **`snorkelai.sdk.client_v3.utils.get`** method.
include_archived	`bool`	`False`	Whether to include archived benchmarks.

Returns

A list of Benchmark objects representing all benchmarks in the given workspace.

Return type

List[Benchmark]

list_criteria

list_criteria(*include_archived=False*)

Retrieves all criteria for this benchmark.

Criteria are the key characteristics that represent the features being optimized for evaluation. Each criteria defines what aspect of the model’s performance is being measured, such as accuracy, relevance, or safety.

Each Criteria object contains:

- criteria_uid: The unique identifier for this criteria.
- benchmark_uid: The ID of the parent benchmark.
- name: The name of the criteria.
- description: A detailed description of what the criteria measures.
- requires_rationale: Whether the criteria requires a rationale explanation.
- label_map: A dictionary mapping user-friendly labels to numeric values.

Parameters

Name	Type	Default	Info
include_archived	`bool`	`False`	Whether to include archived criteria.

Returns

A list of Criteria objects representing all criteria in this benchmark.

Return type

List[Criteria]

Example

Example 1

Get all criteria for a benchmark and list them:

```
benchmark = Benchmark.get(100)
criteria_list = benchmark.list_criteria()
for criteria in criteria_list:
    print(f"Criteria: {criteria.name} - {criteria.description}")
```

list_executions

list_executions(*include_archived=False*)

Retrieves all benchmark executions for this benchmark.

A benchmark execution represents a single run of a benchmark against a dataset, capturing the results and metadata of that evaluation. Executions are returned in chronological order, with the most recent execution last.

Each BenchmarkExecution object contains:

- benchmark_uid: The ID of the parent benchmark.
- benchmark_execution_uid: The unique identifier for this execution.
- name: The name of the execution.
- created_at: Timestamp when the execution was created.
- created_by: Username of the execution creator.
- archived: Whether the execution is archived.

After retrieving executions, you can export their results using `export_latest_execution()` or export the benchmark configuration using `export_config()`. For more information about exporting benchmarks, see Export evaluation benchmark.

Parameters

Name	Type	Default	Info
include_archived	`bool`	`False`	Whether to include archived executions.

Return type

List[BenchmarkExecution]

Example

Example 1

Get all executions for a benchmark and list them:

```
benchmark = Benchmark.get(100)
executions = benchmark.list_executions()
```

update

update(*name=None, description=None, archived=None*)

Updates the benchmark with the given parameters. If a parameter is not provided or is None, the existing value will be left unchanged.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new name of the benchmark.
description	`Optional[str]`	`None`	The new description of the benchmark.
archived	`Optional[bool]`	`None`	Whether the benchmark should be archived.

Return type

`None`

Example

```
benchmark = Benchmark.get(100)
benchmark.update(name="New Name", description="New description")
```

property **archived***: bool*

Return whether the benchmark is archived *property* **benchmark_uid***: int*

Return the UID of the benchmark

property **created_at***: datetime*

Return the timestamp when the benchmark was created

property **description***: str | None*

Return the description of the benchmark

property **name***: str*

Return the name of the benchmark

property **uid***: int*

Return the UID of the benchmark

property **updated_at***: datetime*

Return the timestamp when the benchmark was last updated

snorkelai.sdk.develop.BenchmarkExecution

final class **snorkelai.sdk.develop.BenchmarkExecution**(*benchmark_uid, benchmark_execution_uid, name, created_at, created_by, archived*)

Bases: `Base`

Represents a single execution run of a benchmark for a dataset.

A benchmark execution exports comprehensive evaluation data including per-datapoint scores (evaluator outputs, rationales, and ground truth agreement), slice membership, benchmark and execution metadata, including timing information and execution context.

__init__

__init__(*benchmark_uid, benchmark_execution_uid, name, created_at, created_by, archived*)

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the parent Benchmark. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/10` indicates a benchmark with `benchmark_uid` of `100`.
benchmark_execution_uid	`int`		The unique identifier for this execution.
name	`str`		The name of the execution.
created_at	`datetime`		Timestamp of when this execution was run.

created_by	`str`		Username of the user who ra this execution.
archived	`bool`		Whether this execution is archived.

Methods

`__init__`(benchmark_uid, ...)	
`create`(benchmark_uid[, name, criteria_uids, ...])	Create a benchmark execution.
`delete`(benchmark_uid, benchmark_execution_uid)	Delete (archive) a benchmark execution.
`export`(filepath[, config, connector_config_uid])	Export information associated with this benchmark execution.
`get`(benchmark_uid, benchmark_execution_uid)	Get a benchmark execution by its unique identifier.
`list`(benchmark_uid[, include_archived])	List all benchmark executions for a given benchmark.
`update`(archived)	Update the state of the benchmark execution.

Attributes

`archived`	Return whether the benchmark execution is archived
`benchmark_execution_uid`	Return the UID of the benchmark execution
`benchmark_uid`	Return the UID of the parent benchmark
`created_at`	Return the timestamp when the benchmark execution was created
`created_by`	Return the username of the user who created the benchmark execution
`name`	Return the name of the benchmark execution

`uid`	Return the UID of the benchmark execution

create

classmethod **create**(*benchmark_uid, name=None, criteria_uids=None, datasource_uids=None, splits=None*)

Create a benchmark execution.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark to create an execution for.
name	`Optional[str]`	`None`	Optional name for the benchmark execution.
criteria_uids	`Optional[List[int]]`	`None`	List of criteria UIDs to include in the execution.
datasource_uids	`Optional[List[int]]`	`None`	List of datasource UIDs to include in the execution.
splits	`Optional[List[str]]`	`None`	List of splits to include in the execution.

Returns

The created benchmark execution.

Return type

BenchmarkExecution

Example

```
from snorkelai.sdk.develop import BenchmarkExecution
BenchmarkExecution.create(benchmark_uid=123, name="Test
Execution",datasource_uids=[1, 2, 3], splits=["train", "test"])
```

delete

classmethod **delete**(*benchmark_uid, benchmark_execution_uid*)

Delete (archive) a benchmark execution.

This performs a soft delete by archiving the benchmark execution. Hard deletion is not supported.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark.
benchmark_execution_uid	`int`		The unique identifier of the benchmark execution to delete.

Raises

ValueError – If the benchmark execution is not found.

Return type

`None`

Example

```
from snorkelai.sdk.develop import BenchmarkExecution
BenchmarkExecution.delete(benchmark_uid=123,
benchmark_execution_uid=456)
```

export

export(*filepath, config=None, connector_config_uid=None*)

Export information associated with this benchmark execution. The exported data includes:

- Benchmark metadata for the associated benchmark
- Execution metadata for this execution
- Each datapoint lists its evaluation score, which includes:
  - The evaluator outputs
  - Rationale
  - Agreement with ground truth
- Each datapoint lists its slice membership(s)
- (CSV exports only) Uploaded user columns and ground truth

The export includes all datapoints without filtering or sampling. Some datapoints may have missing evaluation scores if the benchmark was not executed against them (for example, datapoints in the test split).

Parameters

Name	Type	Default	Info
filepath	`str`		The filepath wl you want to wr exported data.
config	`Union[JsonExportConfig, CsvExportConfig, None]`	`None`	A `JsonExportC` or `CsvExportCo` object. Default JSON. No addit configuration i required for JS

			exports. For CS exports, the following para are supported • `sep`: The separator between co Default is `,` • `quotechar` character u quote fields Default is `"` • `escapecha` The charact used to esca special characters. Default is `\`
connector_config_uid	`Optional[int]`	`None`	Optional UID c connector con use for the exp **Required** only export destina a remote, priva bucket (a priva or GCS bucket requires crede **Ignored** if the e destination is a public bucket (public S3 or GC bucket that do require creden or if the export destination is a file.

Return type

`None`

Examples

Example 1

Export a benchmark execution to a local file:

```
from snorkelai.sdk.develop import Benchmark

benchmark = Benchmark.get(100)
execution = benchmark.list_executions()[0]
execution.export("benchmark_execution.json")
```

Example 2

Export a benchmark execution to a S3 bucket using a connector config:

```
from snorkelai.sdk.develop import Benchmark

benchmark = Benchmark.get(100)
execution = benchmark.list_executions()[0]
execution.export("s3://MY-BUCKET/MY-PATH/benchmark_execution.json",
connector_config_uid=1)
```

get

classmethod **get**(*benchmark_uid, benchmark_execution_uid*)

Get a benchmark execution by its unique identifier.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark.
benchmark_execution_uid	`int`		The unique identifier of the benchmark execution.

Returns

The requested benchmark execution.

Return type

BenchmarkExecution

Raises

ValueError – If the benchmark execution is not found.

Example

```
from snorkelai.sdk.develop import BenchmarkExecution

BenchmarkExecution.get(benchmark_uid=123,
benchmark_execution_uid=456)
```

list

static **list**(*benchmark_uid, include_archived=False*)

List all benchmark executions for a given benchmark.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the parent Benchmark. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.

include_archived	`bool`	`False`	Whether to include archived executions. Defaults to False.

Return type

`List`[`BenchmarkExecution`]

update

update(*archived*)

Update the state of the benchmark execution.

Parameters

Name	Type	Default	Info
archived	`bool`		Whether the benchmark execution should be archived.

Return type

`None`

property **archived**: *bool*

Return whether the benchmark execution is archived

property **benchmark_execution_uid**: *int*

Return the UID of the benchmark execution

property **benchmark_uid**: *int*

Return the UID of the parent benchmark

property **created_at**: *datetime*

Return the timestamp when the benchmark execution was created

property **created_by**: *str*

Return the username of the user who created the benchmark execution

property **name**: *str*

Return the name of the benchmark execution

property **uid**: *int*

Return the UID of the benchmark execution

snorkelai.sdk.develop.Cluster

final class snorkelai.sdk.develop.Cluster(*cluster_uid, error_analysis_uid, name, description=None, improvement_strategy=None, examples=None, datapoint_count=0, virtualized_dataset_uid=None, created_at=datetime.datetime(2026, 1, 28, 1, 59, 39, 168255), updated_at=datetime.datetime(2026, 1, 28, 1, 59, 39, 168258)*)

Bases: `Base`

Provides methods for viewing and updating clusters and the ability to view datapoints assigned to a cluster.

Clusters represent groups of similar datapoints identified during error analysis. They help identify common failure patterns in model predictions and provide insights for targeted improvements. Clusters can currently only be created and deleted through the ErrorAnalysis class.

Read more in the Error Analysis Guide.

Using the `Cluster` class requires the following import:

```
from snorkelai.sdk.develop import Cluster
```

__init__

__init__(*cluster_uid, error_analysis_uid, name, description=None, improvement_strategy=None, examples=None, datapoint_count=0, virtualized_dataset_uid=None, created_at=datetime.datetime(2026, 1, 28, 1, 59, 39, 168255), updated_at=datetime.datetime(2026, 1, 28, 1, 59, 39, 168258)*)

Initializes a Cluster instance.

Parameters

Name	Type	Default
cluster_uid	`int`	
error_analysis_uid	`int`	

name	`str`	
description	`Optional[str]`	`None`
improvement_strategy	`Optional[str]`	`None`
examples	`Optional[List[str]]`	`None`
datapoint_count	`int`	`0`
virtualized_dataset_uid	`Optional[int]`	`None`
created_at	`Optional[datetime]`	`datetime.datetime(202 1, 28, 1, 59, 39, 168255)`

updated_at	`Optional[datetime]`	`datetime.datetime(202 1, 28, 1, 59, 39, 168258)`

Methods

`__init__`(cluster_uid, error_analysis_uid, name)	Initializes a Cluster instance.
`create`()	Creates this cluster.
`delete`(cluster_uid)	Deletes this cluster.
`get`(cluster_uid)	Retrieves an existing cluster by its unique identifier.
`get_cluster_membership`()	Fetches datapoint membership for a specific cluster.
`get_clusters`(error_analysis_uid, benchmark_uid)	Fetches clusters from a completed error analysis.
`update`([name, description])	Updates the cluster properties.

Attributes

`created_at`	The timestamp when the cluster was created.
`datapoint_count`	The number of datapoints in the cluster.
`description`	The description of the cluster.
`error_analysis_uid`	The unique identifier for the associated error analysis run.
`examples`	Example datapoints in the cluster.
`improvement_strategy`	The suggested improvement strategy for the cluster.
`name`	The name of the cluster.
`uid`	The unique identifier for the cluster.

`updated_at`	The timestamp when the cluster was last updated.
`virtualized_dataset_uid`	The unique identifier for the virtualized dataset containing the datapoints in the cluster.

create

classmethod **create**()

Creates this cluster.

Raises

NotImplementedError – Cluster creation is not supported directly. Use ErrorAnalysis to create clusters.

Return type

`Cluster`

delete

classmethod **delete**(*cluster_uid*)

Deletes this cluster.

Parameters

Name	Type	Default	Info
cluster_uid	`int`		Unique identifier of the cluster to delete.

Raises

NotImplementedError – Cluster deletion is not supported. Delete the associated error analysis run to remove clusters.

Return type

`None`

get

classmethod **get**(*cluster_uid*)

Retrieves an existing cluster by its unique identifier.

Parameters

Name	Type	Default	Info
cluster_uid	`int`		Unique identifier of the cluster to retrieve.

Returns

The Cluster instance for the specified cluster.

Return type

Cluster

Raises

ValueError – If no cluster exists with the given ID.

Example

```
from snorkelai.sdk.develop import Cluster
cluster = Cluster.get(cluster_uid=123)
```

get_cluster_membership

get_cluster_membership()

Fetches datapoint membership for a specific cluster.

Returns

DataFrame containing all the datapoints in the cluster.

Return type

`pd.DataFrame`

Raises

ValueError – If there are no datapoints assigned to the cluster.

Example

```
from snorkelai.sdk.develop import Cluster
cluster = Cluster.get(cluster_uid=123)
membership_df = cluster.get_cluster_membership()
```

get_clusters

classmethod **get_clusters**(*error_analysis_uid, benchmark_uid*)

Fetches clusters from a completed error analysis.

Parameters

Name	Type	Default	Info
error_analysis_uid	`int`		Unique identifier of the error analysis run.
benchmark_uid	`int`		Unique identifier of the benchmark associated with the error analysis run.

Returns

List of clusters.

Return type

List[Cluster]

Raises

- **RuntimeError** – If called before analysis is complete.
- **ValueError** – If analysis failed or was deleted.

Example

```
from snorkelai.sdk.develop import Cluster
clusters = Cluster.get_clusters(error_analysis_uid=123,
benchmark_uid=456)
```

update

update(*name=None, description=None*)

Updates the cluster properties.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new name for the cluster, by default None.
description	`Optional[str]`	`None`	The new description for the cluster, by default None.

Raises

ValueError – If there are other errors during cluster update.

Return type

`None`

Example

```
from snorkelai.sdk.develop import Cluster
cluster = Cluster.get(cluster_uid=123)
cluster.update(name="New cluster name", description="Updated
description")
```

property **created_at**: *datetime*

The timestamp when the cluster was created.

property **datapoint_count**: *int*

The number of datapoints in the cluster.

property **description**: *str | None*

The description of the cluster.

property **error_analysis_uid**: *int*

The unique identifier for the associated error analysis run.

property **examples**: *List[str] | None*

Example datapoints in the cluster.

property **improvement_strategy**: *str | None*

The suggested improvement strategy for the cluster.

property **name**: *str*

The name of the cluster.

property **uid**: *int*

The unique identifier for the cluster.

property **updated_at**: *datetime*

The timestamp when the cluster was last updated.

property **virtualized_dataset_uid**: *int | None*

The unique identifier for the virtualized dataset containing the datapoints in the cluster.

snorkelai.sdk.develop.CodeEvaluator

final class snorkelai.sdk.develop.CodeEvaluator(*benchmark_uid, criteria_uid, evaluator_uid*)

Bases: `Evaluator`

An evaluator that uses custom Python code to assess an AI application's responses.

A code evaluator uses custom Python functions to evaluate datapoints containing AI application responses, categorizing them into one of a criteria's labels by assigning the corresponding integer score and optional rationale. The evaluator function takes a datapoint as input and returns a score based on the criteria's label schema.

The evaluation function can implement any Python logic needed to assess the AI application's response.

Read more in the Evaluation overview.

Using the `CodeEvaluator` class requires the following import:

```
from snorkelai.sdk.develop import CodeEvaluator
```

Examples

Example 1

Creates a new code evaluator, assessing the length of the AI application's response:

```
import pandas as pd

def evaluate(df: pd.DataFrame) -> pd.DataFrame:
    results = pd.DataFrame(index=df.index)
    results["score"] = df["response"].str.len() > 10  # Simple length
check
    results["rationale"] = "Response length evaluation"
    return results

# Create a new code evaluator
evaluator = CodeEvaluator.create(
    criteria_uid=100,
    evaluate_function=evaluate,
    version_name="Version 1"
)
```

Example 2

Gets an existing code evaluator:

```
# Get existing evaluator
evaluator = CodeEvaluator.get(
    evaluator_uid=300,
)
```

__init__

__init__(*benchmark_uid, criteria_uid, evaluator_uid*)

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark that contains the criteria. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
evaluator_uid	`int`		The unique identifier for this evaluator.

Methods

`__init__`(benchmark_uid, criteria_uid, ...)	
`create`(criteria_uid, evaluate_function[, ...])	Creates a new code evaluator for a criteria.
`delete`(evaluator_uid)	Deletion of an evaluator is not implemented.
`execute`(split[, num_rows, version_name, sync])	Executes the code evaluator against a dataset split.
`get`(evaluator_uid)	Retrieves a code evaluator for a given uid.

`get_execution_result`(execution_uid)	Retrieves the evaluation results for a specific evaluation execution.
`get_executions`()	Retrieves all executions for this code evaluator.
`get_versions`()	Retrieves all code version names for this code evaluator.
`poll_execution_result`(execution_uid[, sync])	Polls the job status and retrieves partial results.
`update`([version_name, evaluate_function])	Updates the code evaluator with a new evaluation function.

Attributes

`benchmark_uid`	Return the UID of the parent benchmark
`criteria_uid`	Return the UID of the parent criteria
`evaluator_uid`	Return the UID of the evaluator
`uid`	Return the UID of the evaluator

create

classmethod **create**(*criteria_uid, evaluate_function, version_name=None*)

Creates a new code evaluator for a criteria.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
evaluate_function	`Callable[[DataFrame], DataFrame]`		A Python function that performs the

			evaluation. This function must: • Be named `evaluate` • Accept a pandas DataFrame as input • Return a pandas DataFrame as output The input DataFrame has a MultiIndex with a single level named `__DATAPOINT_UID` that holds the unique identifier of the datapoint. Values in the index are of the form `("uid1",)`. The output DataFrame must: • Have the same index as the input DataFrame • Include a column named `score` containing the evaluation results • Optionally include a column named `rationale` with

			explanations for the scores
version_name	`Optional[str]`	`None`	The name for the initial code version. If not provided, a default name will be generated.

Raises

ValueError – If the function name is not `evaluate` or if `evaluate_function` is not callable.

Return type

`CodeEvaluator`

Example

Example 1

Creates a new code evaluator, assessing the length of the AI application's response:

```python
import pandas as pd

def evaluate(df: pd.DataFrame) -> pd.DataFrame:
    results = pd.DataFrame(index=df.index)
    results["score"] = df["response"].str.len() > 10
    results["rationale"] = "Response length evaluation"
    return results

evaluator = CodeEvaluator.create(
    criteria_uid=100,
    evaluate_function=evaluate,
)
```

execute

execute(*split, num_rows=None, version_name=None, sync=False*)

Executes the code evaluator against a dataset split.

This method runs the evaluation code against the specified dataset split. If no version name is specified, it uses the latest version.

Parameters

Name	Type	Default	Info
split	`str`		The dataset split to evaluate against (e.g., “train”, “test”, “validation”).
num_rows	`Optional[int]`	`None`	The number of rows to evaluate. If `None`, evaluates all rows in the split.
version_name	`Optional[str]`	`None`	The code version name to run. If `None`, the latest code version is used.
sync	`bool`	`False`	Whether to wait for the job to complete. If `True`, blocks until completion.

Return type

`int`

Example

Example 1

Run the latest code version and poll for results:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(split="test", num_rows=100)
status, results =
evaluator.poll_execution_result(code_execution_uid, sync=False)
```

Example 2

Run a specific code version:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(
    split="test",
    num_rows=100,
    version_name="v1.0"
)
```

get

classmethod **get**(*evaluator_uid*)

Retrieves a code evaluator for a given uid.

Parameters

Name	Type	Default	Info
evaluator_uid	`int`		The unique identifier for the evaluator.

Returns

A CodeEvaluator instance.

Return type

CodeEvaluator

Raises

ValueError – If the evaluator with the given uid is not a CodeEvaluator.

get_execution_result

get_execution_result(*execution_uid*)

Retrieves the evaluation results for a specific evaluation execution.

This method reads the evaluation results from the database for the given evaluation execution UID.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The evaluation execution UID to get results for.

Return type

`Dict`[`str`, `Dict`[`str`, `Union`[`str`, `int`, `float`, `bool`]]]

Example

Example 1

Get the results of a code execution:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(split="test", num_rows=100)
results = evaluator.get_execution_result(code_execution_uid)
print(f"Evaluation scores: {results}")
```

get_executions

get_executions()

Retrieves all executions for this code evaluator.

This method fetches all executions that have been run using this evaluator. Executions are returned in chronological order, with the oldest execution first.

The dictionary contains the following keys: :rtype: `List`[`Dict`[`str`, `Any`]]

- `execution_uid`: The execution UID
- `created_at`: The timestamp when the execution was created
- `code_version_name`: The name of the code version used for the execution

Example

Example 1

Get all executions for an evaluator:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
executions = evaluator.get_executions()
for execution in executions:
    print(f"Execution {execution['execution_uid']}:
{execution['created_at']}")
```

get_versions

get_versions()

Retrieves all code version names for this code evaluator.

This method fetches all code version names that have been created for this evaluator. Versions are returned in chronological order, with the oldest version first.

Return type

`List[str]`

Example

Example 1

Get all code version names for an evaluator:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
versions = evaluator.get_versions()
for version in versions:
    print(f"Version: {version}")
```

poll_execution_result

poll_execution_result(*execution_uid, sync=False*)

Polls the job status and retrieves partial results.

This method checks the current status of the evaluation job and returns both the job status and any available results. The current status can be `running`, `completed`, `failed`, `cancelled`, or `unknown`.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The code execution UID to poll for.
sync	`bool`	`False`	Whether to wait for the job to complete. If `False`, returns immediately with current status and partial results.

Return type

`Tuple`[`str`, `Dict`[`str`, `Dict`[`str`, `Union`[`str`, `int`, `float`, `bool`]]]]

Example

Example 1

Poll for job status and partial results:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(split="test", num_rows=100)
status, results =
evaluator.poll_execution_result(code_execution_uid, sync=False)
print(f"Job status: {status}")
if results:
    print(f"Partial results: {results}")
```

update

update(*version_name=None, evaluate_function=None*)

Updates the code evaluator with a new evaluation function.

This method creates a new code version containing the provided evaluation function. The function must be declared with the name `evaluate` and will be used to assess datapoints containing AI application responses against the criteria.

Parameters

Name	Type	Default	I
version_name	`Optional[str]`	`None`	The nam new code not provi default n be gener
evaluate_function	`Optional[Callable[[DataFrame], DataFrame]]`	`None`	A Python that perf evaluatio function • Be nai `evalu` • Accep DataF input • Returi DataF outpu The inpu DataFran MultiInde single lev `__DATAF` that hold unique ic the datap Values in are of the `("uid1"` The outp DataFran • Have t index

			input • Includ name contai evalua result • Optior includ name `ratic` with explar the sc

Raises

ValueError – If the function name is not ‘evaluate’ or if evaluate_function is not provided.

Return type

`None`

Example

Example 1

Update a code evaluator with a new evaluation function:

```python
import pandas as pd

def evaluate(df: pd.DataFrame) -> pd.DataFrame:
    results = pd.DataFrame(index=df.index)

    # Add random scores between 0 and 1
    results["score"] = np.random.randint(0, 3, size=len(df))

    # Add random rationales
    rationale_options = [
        "This response is accurate and relevant.",
        "The answer demonstrates good understanding.",
        "Response shows appropriate reasoning.",
        "This is a well-formed answer.",
        "The content is factually correct.",
    ]
    results["rationale"] = np.random.choice(rationale_options,
size=len(df))
    return results

evaluator = CodeEvaluator.get(evaluator_uid=300)
version_name = evaluator.update("v2.0", evaluate_function=evaluate)
```

snorkelai.sdk.develop.Criteria

final class snorkelai.sdk.develop.Criteria(*benchmark_uid, criteria_uid, name, metric_label_schema_uid, description=None, rationale_label_schema_uid=None, archived=False*)

Bases: `Base`

A criteria represents a specific characteristic or feature being evaluated as part of a benchmark.

Criteria define what aspects of a model or AI application's performance are being measured, such as accuracy, relevance, safety, and other qualities. Each criteria is associated with a benchmark and has an evaluator that assesses whether a model's output satisfies that criteria.

The heart of each criteria is its associated label schema, which defines what, exactly, the criteria is measuring, and maps each option to an integer.

For example, a criteria that measures accuracy might have a label schema that defines the following labels:

- `INCORRECT`: 0
- `CORRECT`: 1

A criteria that measures readability might have a label schema that defines the following labels:

- `POOR`: 0
- `ACCEPTABLE`: 1
- `EXCELLENT`: 2

Read more in the Evaluation overview.

__init__

__init__(*benchmark_uid, criteria_uid, name, metric_label_schema_uid, description=None, rationale_label_schema_uid=None, archived=False*)

Parameters

Name	Type	Default	Info

benchmark_uid	`int`		The unique identifier parent Benchmark. T `benchmark_uid` is v the URL of the bench page in the Snorkel G example, `https://Y SNORKEL-INSTANCE/benchma` indicates a benchma `benchmark_uid` of
criteria_uid	`int`		The unique identifier criteria.
name	`str`		The name of the crite
metric_label_schema_uid	`int`		The ID of the schema the metric labels.
description	`Optional[str]`	`None`	A detailed descriptio the criteria measures
rationale_label_schema_uid	`Optional[int]`	`None`	The ID of the schema rationale labels (if ap
archived	`bool`	`False`	Whether the criteria archived.

Examples

Using the `Criteria` class requires the following import:

```
from snorkelai.sdk.develop import Criteria
```

Create a new criteria:

```
# Create a new criteria
criteria = Criteria.create(
    benchmark_uid=100,
    name="Accuracy",
    description="Measures response accuracy",
    label_map={"Correct": 1, "Incorrect": 0},
    requires_rationale=True
)
```

Get an existing criteria:

```
# Get existing criteria
criteria = Criteria.get(criteria_uid=100)
```

Methods

`__init__`(benchmark_uid, criteria_uid, name, ...)	
`archive`()	Archives the criteria, hiding it from the UI and Benchmark.list_criteria method.
`create`(benchmark_uid, name, label_map[, ...])	Create a new criteria for a benchmark.
`delete`(criteria_uid)	Deletion of a criteria is not implemented.
`get`(criteria_uid)	Get an existing criteria by its UID.
`get_evaluator`()	Retrieves the evaluator associated with this criteria.
`update`([name, description, archived])	Updates the criteria with the given parameters.

Attributes

`archived`	Return whether the criteria is archived
`benchmark_uid`	Return the UID of the parent benchmark
`criteria_uid`	Return the UID of the criteria
`description`	Return the description of the criteria
`metric_label_schema_uid`	Return the UID of the metric label schema

`name`	Return the name of the criteria
`rationale_label_schema_uid`	Return the UID of the rationale label schema
`uid`	Return the UID of the criteria

archive

archive()

Archives the criteria, hiding it from the UI and Benchmark.list_criteria method.

Use `snorkelai.sdk.develop.benchmarks.Benchmark.list_criteria()` with `include_archived=True` to view archived criteria.

Return type

`None`

create

static create(*benchmark_uid, name, label_map, description=None, requires_rationale=False*)

Create a new criteria for a benchmark.

Your `label_map` must use consecutive integers starting from `0`. For example, if you have three labels, you must use the values `0`, `1`, and `2`.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the parent Benchmark.
name	`str`		The name of the criteria.
label_map	`Dict[str, int]`		A dictionary mapping user-friendly labels to numeric values. The key "UNKNOWN" will always be added with value -1.

			Dictionary values must be consecutive integers starting from 0.
description	`Optional[str]`	`None`	A detailed description of what the criteria measures.
requires_rationale	`bool`	`False`	Whether the criteria requires rationale.

Returns

A new Criteria object representing the created criteria.

Return type

Criteria

Raises

ValueError – If label_map is empty or has invalid values.

Example

```
criteria = Criteria.create(
    benchmark_uid=200,
    name="Accuracy",
    description="Measures response accuracy",
    label_map={"Correct": 1, "Incorrect": 0},
    requires_rationale=True
)
```

delete

classmethod **delete**(*criteria_uid*)

Deletion of a criteria is not implemented.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier of the criteria.

Return type

`None`

get

static **get**(*criteria_uid*)

Get an existing criteria by its UID.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier for the criteria.

Returns

A Criteria object representing the existing criteria.

Return type

Criteria

Raises

ValueError – If the criteria is not found.

Example

```
criteria = Criteria.get(criteria_uid=100)
```

get_evaluator

get_evaluator()

Retrieves the evaluator associated with this criteria.

An evaluator is a prompt or code snippet that assesses whether a model's output satisfies the criteria. Each criteria has one evaluator that assesses each datapoint against the criteria's label schema and chooses the most appropriate label, in the form of the associated integer.

The evaluator can be either a code evaluator (using custom Python functions) or a prompt evaluator (using LLM prompts).

Raises

IndexError – If no evaluator is found for this criteria.

Return type

`Evaluator`

Example

Example 1

Get the evaluator for a criteria and check its type:

```
from snorkelai.sdk.develop import PromptEvaluator, CodeEvaluator

criteria = Criteria.get(criteria_uid=100)
evaluator = criteria.get_evaluator()

if isinstance(evaluator, CodeEvaluator):
    print("This is a code evaluator")
elif isinstance(evaluator, PromptEvaluator):
    print("This is a prompt evaluator")
```

update

update(*name=None, description=None, archived=None*)

Updates the criteria with the given parameters. If a parameter is not provided or is None, the existing value will be left unchanged.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The name of the criteria.
description	`Optional[str]`	`None`	A detailed description of what the criteria measures.
archived	`Optional[bool]`	`None`	Whether the criteria is archived.

Return type

`None`

Example

```
criteria = Criteria.get(criteria_uid=100)
criteria.update(name="New Name", description="New description")
```

property **archived**: *bool*

Return whether the criteria is archived

property **benchmark_uid**: *int*

Return the UID of the parent benchmark

property **criteria_uid**: *int*

Return the UID of the criteria

property **description**: *str | None*

Return the description of the criteria

property **metric_label_schema_uid**: *int*

Return the UID of the metric label schema

property **name**: *str*

Return the name of the criteria

property **rationale_label_schema_uid**: *int | None*

Return the UID of the rationale label schema

property **uid**: *int*

Return the UID of the criteria

snorkelai.sdk.develop.CsvExportConfig

class snorkelai.sdk.develop.CsvExportConfig(*sep=',', quotechar='"', escapechar='\\\\'*)

Bases: `object`

Benchmark execution CSV export configuration

__init__

__init__(*sep=',', quotechar='"', escapechar='\\\\'*)

Parameters

Name	Type	Default	Info
sep	`str`	`','`	The separator between columns.
quotechar	`str`	`'"'`	The character used to quote fields.
escapechar	`str`	`'\\\\'`	The character used to escape special characters.

Methods

`__init__`([sep, quotechar, escapechar])	

snorkelai.sdk.develop.Dataset

final class **snorkelai.sdk.develop.Dataset**(*name, uid, mta_enabled*)

Bases: `Base`

The Dataset object represents a dataset in Snorkel Flow.

Datasets Quickstart

In this quickstart, we will create a Dataset and upload a file to that Dataset as a data source. We will then show how you might go about ingesting that data into the platform.

We will need the following imports

```
from snorkelai.sdk.develop import Dataset
import snorkelai.sdk.client as sai
import pandas as pd
ctx = sai.SnorkelSDKContext.from_endpoint_url()
```

We will begin by creating a new Dataset.

```
>>> contracts_dataset = Dataset.create("contracts-dataset")
Successfully created dataset contracts-dataset with UID 0 in workspace 0
```

Next, we will attempt to save a file to the Dataset as a data source. This file will be in S3. File upload will initially fail because this file contains null values.

```
>>> contracts_dataset.create_datasource("s3://snorkel-contracts-
dataset/dev.parquet", uid_col="uid", split="train")
UserInputError: Errors...
```

In this particular example, we decide we don't care about these rows, so we can use Pandas to edit the file and remove the null values. We can then re-upload the data, this time uploading the DataFrame directly without needing to save it to a file again. In some other cases, you may want to either edit those null cells or fix them in your upstream data pipeline.

```
>>> df = pd.read_parquet("s3://snorkel-contracts-dataset/dev.parquet")
>>> df = df.dropna()
>>> contracts_dataset.create_datasource(df, uid_col="uid",
split="train")
+0.07s Starting data ingestion
+1.85s Ingesting data
+2.05s Data ingestion complete
```

To verify that has worked correctly, we can view this Dataset’s data sources.

```
>>> contracts_dataset.datasources
[{'datasource_uid': 668,...}]
```

Dataset Concepts

Datasets

Datasets are how your data is represented in Snorkel Flow. Snorkel Flow projects always begin with a single Dataset. Datasets bring external data into Snorkel Flow and help manage that data once it has been ingested. Datasets are composed of individual chunks of data, called **data sources**, and provides an interface for managing individual data sources.

Data Sources

Data sources are the individual chunks of data that make up a Dataset. A data source can be a file you upload from local storage, a file located in a remote (S3, MinIO, etc.) storage service, or an in-memory Pandas DataFrame. Data sources shouldn’t be touched directly, but should be managed by interfacing with their parent Dataset. The best way to deal with data sources is to treat them as blocks of data, which can be added and removed but only occasionally changed. Data sources can be given names during their creation, but are usually referred to using a data source UID, an integer ID assigned to each data source when it is created.

Derived Data Sources

When an application is created using a dataset, Snorkel Flow will create a *derived* data source for each data source in the dataset. Derived data sources are intermediate representations of data that track the lineage of the data as it is being processed and are associated with only one application. Note that some operations, such as changing the split of a data source, don’t propagate to any of the derived data source once they are derived, and vice versa. Derived data sources are viewable in the Snorkel Flow UI on the “View Data Sources” button, accessible from the “Develop” screen of an application.

Modifying Data

In general, data sources should be treated as immutable. This means that you should avoid modifying the underlying data source once it has been uploaded. If your goal is to filter out rows, add feature columns, or remove feature columns, you should use an Operator to do so. Alternatively, you can modify your data upstream of Snorkel and create a new Dataset with your edited data.

The Python SDK provides limited support for specific one-off operations on data sources. Sometimes you might need to reformat the data in an existing column to make it compatible with processing logic. In this case, you can use the `dataset.update_datasource_data` method to swap out an existing data source for a new one with the updated data. However, be aware that this is an irreversible change, and updating data in this way is an expensive operation that will require all downstream applications to be refreshed.

Splits

Data sources belong to *splits*. Splits help dictate how the data will be used in the model development process. Data sources allocated to the **train** split will be used for model training and labeling function development. Data sources allocated to the **valid** split will be used to validate models iteratively and to perform error analysis. Data sources allocated to the **test** split will be used to evaluate the final model. Data source splits may be updated as needed, but be aware that model metrics and labeling function performance will change based on how the splits are allocated.

Data Upload Guardrails

When you upload data to Snorkel Flow, it must pass a series of safety checks to ensure that the data is valid and safe to load into the platform. These checks include:

> - **Number of rows**: A single data source should not exceed 10 million rows. If your data source exceeds this limit, you should split it into multiple data sources before uploading.
> - **Column memory**: The average memory usage of a single column must be under 20MB across all columns in your data source. For performance, the average column memory usage should be under 5MB. If your data source exceeds this limit, you should split it into multiple data sources before uploading.

- **Null values**: Snorkel Flow will not permit data to be uploaded if any null values exist in that data source. If you have null values in your data, you might want to clean them up with the Pandas `fillna()` method before uploading.
- **Unique integer index**: Snorkel Flow requires that each data source have a unique integer index column. The values in this index must be unique among all datasources in the Dataset. The values must also be unique, non-negative integers. If your Dataset does not already have this stable index column, you must create one before uploading.
- **Consistent schema**: All data sources in a single Dataset should have the same columns. All columns that are in multiple data sources must have the same type. If you have columns that exist in some data sources but not others, you may see unexpected behavior in downstream tasks.

Fetching UIDs

Methods in the `Dataset` class will sometimes require a UID parameter. This is the unique identifier for the Dataset within Snorkel Flow. The Dataset UID can be retrieved by calling `.uid` on a Dataset object. Data source methods will sometimes require a data source UID, which can be retrieved by printing out the datasources by calling `my_dataset.datasources`. The data source UID is the `datasource_uid` field in the returned dictionary.

__init__

__init__(*name, uid, mta_enabled*)

Create a dataset object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods

Parameters

Name	Type	Default	Info
name	`str`		The human-readable name of the dataset. Must be unique within the workspace.

uid	`int`		The unique integer identifier for the dataset within Snorkel Flow.
mta_enabled	`bool`		Whether or not multi-task annotation is enabled for this dataset.

Methods

`__init__`(name, uid, mta_enabled)	Create a dataset object in-memory with necessary properties.
`add_ground_truth`(label_schema_uid, data[, ...])	Add ground truth to a given label schema.
`create`(dataset_name[, enable_mta])	Creates and registers a new Dataset object.
`create_batches`([name, assignees, ...])	Create annotation batches for this dataset.
`create_datasource`(data[, uid_col, name, ...])	Creates a new data source withing the Dataset from either a filepath or a Pandas DataFrame.
`create_label_schema`(name, data_type, ...[, ...])	Create a label schema associated with this dataset.
`delete`(dataset[, force])	Delete a dataset based on the provided identifier
`delete_datasource`(datasource_uid[, force, sync])	Delete a data source.
`get`(dataset)	Fetches an already-created Dataset from Snorkel Flow and returns a Dataset object that can be used to interact with files and data
`get_dataframe`([split, max_rows, ...])	Read the Dataset's data into an in-memory Pandas DataFrame.
`get_ground_truth`(label_schema_uid[, user_format])	Get the ground truth for a given label schema.
`list`()	Get a list of all Datasets.
`update`([name])	Update the metadata for this dataset.

`update_datasource_data`(old_datasource_uid, ...)	This function allows you to replace the data of an existing data source with new data.
`update_datasource_split`(datasource_uid, split)	Change the split of a data source that has already been uploaded to the dataset.

Attributes

`batches`	A list of batches belonging to this Dataset.
`datasources`	A list of data sources and associated metadata belonging to this Dataset.
`label_schemas`	A list of label schemas belonging to this Dataset.
`mta_enabled`	Whether or not multi-task annotation is enabled for this dataset.
`name`	The human-readable name of the dataset.
`uid`	The unique integer identifier for the dataset within Snorkel Flow.

add_ground_truth

add_ground_truth(*label_schema_uid, data, user_format=True, sync=True*)

Add ground truth to a given label schema.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> from snorkelai.sdk.client_v3.utils import poll_job_status
>>> my_dataset = Dataset.get("my-dataset")
>>> df = pd.DataFrame(
>>>     {
>>>         "label": ["POSITIVE", "NEGATIVE", "NEGATIVE",
"POSITIVE"],
>>>         "__DATAPOINT_UID": ["doc::1", "doc::2", "doc::3",
"doc::4"],
>>>     }
>>> )
>>> job_uid = my_dataset.add_ground_truth(
>>>     label_schema_uid=1,
>>>      data=df,
>>>      user_format=True,
>>>      sync=False)
>>> poll_job_status(job_uid)
```

Parameters

Name	Type	Default	Info
label_schema_uid	`int`		The UID of the label schema to add the ground truth to.
data	`DataFrame`		A pandas DataFrame with x_uid as index, and ground truth labels in column 'label'.
user_format	`bool`	`True`	True if ground truth labels in data in user format, otherwise they should be raw label values.
sync	`bool`	`True`	Whether execution should be blocked by this function, by default True.

Returns

The job ID of the ground truth job. None if sync is true.

Return type

`Optional[str]`

create

classmethod **create**(*dataset_name, enable_mta=True*)

Creates and registers a new Dataset object. A Dataset object organizes and collects files and other sources of data for use in Snorkel Flow. A Dataset is restricted to a particular workspace, so only users in that workspace will be able to access that Dataset. Datasets must be initialized with a unique name

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.create(dataset_name="my-dataset")
Successfully created dataset my-dataset with UID 0 in workspace 0
```

Parameters

Name	Type	Default	Info
dataset_name	`str`		A name for the Dataset. This name must be unique within the workspace.
enable_mta	`bool`	`True`	Whether to enable multi-task annotation for this dataset. Enabled by default.

Returns

A Dataset object that can be used to interact with the dataset in Snorkel Flow

Return type

Dataset

create_batches

create_batches(*name=None, assignees=None, label_schemas=None, batch_size=None, randomize=False, random_seed=123, selection_strategy=None, split=None, x_uids=None, filter_by_x_uids_not_in_batch=False, divide_x_uids_evenly_to_assignees=False*)

Create annotation batches for this dataset.

This is the recommended entrypoint for creating batches.

Parameters

Name	Type	Defaul
name	`Optional[str]`	`None`
assignees	`Optional[List[int]]`	`None`
label_schemas	`Optional[List[LabelSchema]]`	`None`
batch_size	`Optional[int]`	`None`
randomize	`Optional[bool]`	`False`
random_seed	`Optional[int]`	`123`

selection_strategy	`Optional[SelectionStrategy]`	`None`
split	`Optional[str]`	`None`
x_uids	`Optional[List[str]]`	`None`
filter_by_x_uids_not_in_batch	`Optional[bool]`	`False`
divide_x_uids_evenly_to_assignees	`Optional[bool]`	`False`

Returns

The list of created batches

Return type

List[Batch]

create_datasource

create_datasource(*data, uid_col=None, name=None, split='train', sync=True, run_checks=True*)

Creates a new data source withing the Dataset from either a filepath or a Pandas DataFrame.

If you provide a filepath: A file can be a CSV or Parquet file that either exists in the local filesystem, or is accessible via an S3-compatible API (such as MinIO, or AWS S3). Files must have a stable integer index column that is unique across all data sources in the dataset.

If you provide a DataFrame: The DataFrame must have a unique integer column that does not contain duplicates across other sources of data. All DataFrame column names must be strings.

The data must pass all validation checks to be registered as a valid data source. If a DataFrame fails to pass all data validation checks, the upload will fail and the data source will not be registered.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.create_datasource("my_data.csv", uid_col="id",
split="train")
+0.07s Starting data ingestion
+1.85s Ingesting data
+2.05s Data ingestion complete
1 # UID of the datasource

>>> my_dataset.create_datasource(df, uid_col="id", name="dataframe-
data", split="train")
+0.07s Starting data ingestion
+1.85s Ingesting data
+2.05s Data ingestion complete
1
```

Parameters

Name	Type	Default	Info
data	`Union[str, DataFrame]`		Either: - A path to a file in the local filesystem, or a path to an S3-compatible API, by default None. If filepath is not provided, a DataFrame must be provided instead - A Pandas DataFrame, by default None. If df is not

			provided, a filepath must be provided instead.
uid_col	`Optional[str]`	`None`	Name of the UID column for this data. The values in this column must be unique non-negative integers that are not duplicated across files. If not specified, the UID column will be generated in the server side.
name	`Optional[str]`	`None`	The name to give this data source. If not provided, the name of the file will be used, by default None. Adding a name is strongly recommended when uploading a DataFrame.
split	`str`	`'train'`	The name of the data split this data belongs to, by default Splits.train.
sync	`bool`	`True`	Whether execution should be blocked by this function, by default True. Note that Dataset().datasources may not be updated immediately if sync=False.
run_checks	`bool`	`True`	Whether we should run datasource checks. Recommended for safety, by default True.

Returns

If sync is True, returns the integer UID of the datasource. If sync is False, returns a job ID that can be monitored with `sai.poll_job_id`

Return type

`Union[str, int]`

create_label_schema

create_label_schema(*name, data_type, task_type, label_map, multi_label=False, description=None, label_column=None, label_descriptions=None, primary_field=None, is_text_label=False, allow_overlapping=None*)

Create a label schema associated with this dataset.

This is the recommended entrypoint for creating label schemas.

Parameters

Name	Type	Default	Info
name	`str`		The name of the label schema.
data_type	`str`		The data type of the label schema.
task_type	`str`		The task type of the label schema.
label_map	`Union[Dict[str, int], List[str]]`		A dictionary mapping label names to their integer values, or a list of label names.
multi_label	`bool`	`False`	Whether the label schema is a multi-label schema, by default False.

description	`Optional[str]`	`None`	A description of the label schema, by default None.
label_column	`Optional[str]`	`None`	The name of the column that contains the labels, by default None.
label_descriptions	`Optional[Dict[str, str]]`	`None`	A dictionary mapping label names to their descriptions, by default None.
primary_field	`Optional[str]`	`None`	The primary field of the label schema, by default None.
is_text_label	`bool`	`False`	Whether the label schema is a text label schema, by default False.
allow_overlapping	`Optional[bool]`	`None`	Enable overlapping labels at the same token position. Defaults to None. Sequence tagging only.

Returns

The label schema object

Return type

LabelSchema

delete

classmethod **delete**(*dataset, force=False*)

Delete a dataset based on the provided identifier

The operation will fail if any applications use this Dataset

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> Dataset.delete("my-dataset")
Successfully deleted dataset my-dataset with UID 0.
```

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		Name or UID of the dataset to delete.
force	`bool`	`False`	If True, delete any applications using the Dataset as well.

Return type

`None`

delete_datasource

delete_datasource(*datasource_uid, force=False, sync=True*)

Delete a data source. Calling delete_datasource will fully remove the data source from the dataset.

> WARNING
>
> The operation will not be permitted if any applications are using the data source to avoid breaking downstream applications. If you are sure you want to delete the data source, use the flag `force=True` to override this check. This function may take a while.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.delete_datasource(1)
Successfully deleted datasource with UID 1.
```

Parameters

Name	Type	Default	Info
datasource_uid	`int`		UID of the data source to delete. See all datasources for this dataset by viewing self.datasources.
force	`bool`	`False`	boolean allowing one to force deletion of a datasource even if that datasource has dependent assets (ground truth, annotations, etc), by default false.
sync	`bool`	`True`	Poll job status and block until complete, by default true.

Returns

Optionally returns job_id if sync mode is turned off

Return type

`Optional[str]`

get

classmethod **get**(*dataset*)

Fetches an already-created Dataset from Snorkel Flow and returns a Dataset object that can be used to interact with files and data

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
Successfully retrieved dataset my-dataset with UID 0 in workspace
0.
```

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		Either the name or UID of the dataset. A list of all accessible datasets can be retrieved with `Dataset.list()`

Returns

A Dataset object that can be used to interact with files and data in Snorkel Flow.

Return type

Dataset

get_dataframe

get_dataframe(*split=None, max_rows=10, target_columns=None, datasource_uid=None, use_source_index=True*)

Read the Dataset's data into an in-memory Pandas DataFrame. If only a subset of columns are required, they can be specified with `target_columns`. Note that changes to the DataFrame will not be reflected in the Dataset. To change the actual data in the dataset, you must swap out the relevant data sources.

> NOTE
>
> By default, only 10 rows are read for memory safety. This limit can be increased by setting `max_rows` to a larger value, but this can be computationally intensive and may lead to unstable behavior.

> NOTE
>
> By default, we will return the original index column name the data source was uploaded with. However, certain SDK workflows might require an internal representation of the index column, such as the `snorkelai.sdk.develop.Deployment.execute` function. If you run into issues because of this, run `dataset.get_dataframe` with the `use_source_index` parameter set to `False`.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> df = my_dataset.get_dataframe(target_columns=["a", "b"])
<pd.DataFrame object with 10 rows and columns a, b>
>>> df = my_dataset.get_dataframe(datasource_uid=0, max_rows=None)
<pd.DataFrame object with 100 rows and columns a, b, c>
```

Parameters

Name	Type	Default	Info
split	`Optional[str]`	`None`	The data split to load, b default None (all splits) Other options are "trai "valid", and "test".
max_rows	`Optional[int]`	`10`	The maximum number rows to read, by defaul Use `max_rows=None` t fetch all rows. Warning setting this to a large va can be computationally intensive and may lead unstable behavior.
target_columns	`Optional[List[str]]`	`None`	A list of desired data columns, in case not al

			columns are required, default None.
datasource_uid	`Optional[int]`	`None`	Fetch a dataframe fron particular `datasource_uid`. A li all datasource UIDs car retrieved with `Dataset().datasou` This can't be used with `split`.
use_source_index	`bool`	`True`	If true, returns the inde column that the data s was originally uploaded with. If false, returns th Snorkel Flow internal column name. True by default.

Returns

A Pandas DataFrame object displaying the data in this dataset

Return type

`pd.DataFrame`

get_ground_truth

get_ground_truth(*label_schema_uid, user_format=True*)

Get the ground truth for a given label schema.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> df = my_dataset.get_ground_truth(1, user_format=True)
>>> df
                        label
    __DATAPOINT_UID
    doc::1              POSITIVE
    doc::2              NEGATIVE
    doc::3              NEGATIVE
    doc::4              POSITIVE
```

Parameters

Name	Type	Default	Info
label_schema_uid	`int`		The UID of the label schema to get the ground truth from.
user_format	`bool`	`True`	Return the ground truth in user format if True, otherwise return the raw label values.

Returns

A pandas DataFrame with x_uid as index, and ground truth labels in column 'label'

Return type

`pd.DataFrame`

list

static **list**()

Get a list of all Datasets. The returned list includes the Dataset UID, the Dataset name, and additional metadata used to keep track of the Dataset's properties.

Examples

```
>>> Dataset.list()
[
    {
        "name": "test-csv-str",
        "uid": 116,
        "datasources": []
    },
    ...
]
```

Returns

List of all dataset objects

Return type

List[Dataset]

update

update(*name=''*)

Update the metadata for this dataset. Only updating the name of this Dataset is currently supported. The new name for the dataset must be unique within the workspace.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get(dataset="my-dataset")
>>> my_dataset.update(name="my-new-dataset")
Successfully renamed dataset with UID 0 to my-new-dataset
```

Parameters

Name	Type	Default	Info
name	`str`	`''`	The new name for this dataset.

Return type

`None`

update_datasource_data

update_datasource_data(*old_datasource_uid, new_data, sync=True*)

This function allows you to replace the data of an existing data source with new data. This function can be used if you find an error in an existing value in a data source, or if you need to update values due to changes in your upstream data pipeline. This function requires that all row indexes in the new data source match the row indexes of the old data source. Additionally, all columns must have the same name and the same type.

If your goal is to change the number of columns, the number of rows, or the type of a column, you should consider using an Operator instead.

⚠ WARNING

This is a potentially dangerous operation, and may take a while to run. For safety, this will always run data source checks on the new data source. Applications and models that use the data source being replaced may become temporarily unavailable as computations are re-run over the new data, and might report different behavior. If you are unsure how to use this function, contact a Snorkel representative.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.datasources
[{"datasource_uid": 1, "datasource_name": "test.csv", "split":
"train"}]
>>> df = my_dataset.get_dataframe(datasource_uid=1, max_rows=None)
>>> df
|   |   a |   b |   c         |
|  0|   1 |   0 | bad_path.pdf|
>>> df.iloc[0, 2] = "good_path.pdf"
>>> my_dataset.update_datasource_data(1, df)
Successfully replaced data in datasource with UID 1.
```

Parameters

Name	Type	Default	
old_datasource_uid	`int`		The UID of the data source you w data sources for this dataset by v
new_data	`Union[str, DataFrame]`		Either (1) A path to a file in the loc compatible API, by default None. must be provided instead, or (2) A df is not provided, a filepath mus UIDs of the new data must exactl Use `dataset.get_dataframe(data` to see the existing data.
sync	`bool`	`True`	Poll job status and block until all j

Returns

Returns a Job ID that can be polled if sync is False. Otherwise returns None

Return type

`Optional[str]`

Raises

ValueError – If the data provided is neither a valid file path or a valid Pandas DataFrame

update_datasource_split

update_datasource_split(*datasource_uid, split*)

Change the split of a data source that has already been uploaded to the dataset. This will impact how the data source is used in all future applications.

⚠ WARNING

This will only impact the Dataset’s data source, and not existing derived data sources. To change the split within applications that have already been created, find the node’s derived data source UID by clicking on “Develop” > “View Data Sources” in the Snorkel Flow UI and use the `sai.update_datasource` function.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.datasources
[{"datasource_uid": 1, "datasource_name": "test.csv", "split":
"train"}]
>>> my_dataset.update_datasource_split(1, "train")
[123, 456, 789]
```

Parameters

Name	Type	Default	Info
datasource_uid	`int`		The integer UID corresponding to the data source you wish to update. You can see a list of all data sources for this dataset by viewing self.datasources.
split	`str`		The new split to assign to this data source. Must be one of “train”, “test”, or “valid”.

Returns

Returns a list of model nodes that have been impacted by changing the split.

Return type

`List[int]`

property **batches***: List[Batch]*

A list of batches belonging to this Dataset.

property **datasources***: List[Dict[str, Any]]*

A list of data sources and associated metadata belonging to this Dataset.

property **label_schemas***: List[LabelSchema]*

A list of label schemas belonging to this Dataset.

property **mta_enabled***: bool*

Whether or not multi-task annotation is enabled for this dataset.

property **name***: str*

The human-readable name of the dataset.

property **uid***: int*

The unique integer identifier for the dataset within Snorkel Flow.

snorkelai.sdk.develop.ErrorAnalysis

final class **snorkelai.sdk.develop.ErrorAnalysis**(*provenance, error_analysis_run_uid*)

Bases: `Base`

Provides methods for creating, monitoring, and retrieving results from error analysis clustering runs.

The clustering algorithm identifies patterns in LLM evaluation failures and groups similar errors together. This enables systematic analysis of model performance issues and identification of common failure modes.

Read more in the Error Analysis Guide.

Using the `ErrorAnalysis` class requires the following import:

```
from snorkelai.sdk.develop import ErrorAnalysis
```

__init__

__init__(*provenance, error_analysis_run_uid*)

Initializes an ErrorAnalysis instance.

Parameters

Name	Type	Default	Info
provenance	`Dict[str, int]`		Tracking information that uniquely identifies the inputs used to create this cluster analysis.
error_analysis_run_uid	`int`		Unique identifier for this specific cluster analysis run.

Methods

`__init__`(provenance, error_analysis_run_uid)	Initializes an ErrorAnalysis instance.
`create`(prompt_execution_uid, *[, sync])	Creates and triggers error analysis clustering job for LLM evaluation results.

`delete`(error_analysis_run_uid)	Deletes this error analysis run and all associated cluster data.
`get`(error_analysis_run_uid)	Retrieves an existing error analysis by its unique run identifier.
`get_clusters`()	Fetches clusters from a completed error analysis.
`get_latest`(prompt_execution_uid)	Gets the most recent error analysis run for a specific prompt execution.
`update`()	Update an error analysis run

Attributes

`uid`	Return the UID of the error analysis run

create

classmethod **create**(*prompt_execution_uid, *, sync=False*)

Creates and triggers error analysis clustering job for LLM evaluation results.

Parameters

Name	Type	Default	Info
prompt_execution_uid	`int`		Unique identifier of the prompt execution.
sync	`bool`	`False`	If `True`, method blocks until clustering completes and returns ready ErrorAnalysis. If `False`, returns immediately with ErrorAnalysis that requires waiting to get clusters.

Returns

An ErrorAnalysis instance representing the clustering job. If sync = `False`, get_clusters will fail until analysis is complete. If sync = `True`, results are immediately available.

Return type

ErrorAnalysis

Raises

- **ValueError** – If benchmark, prompt execution or criteria don’t exist or are not valid.
- **ValueError** – If prompt execution has no evaluation results or insufficient error cases to cluster.

Examples

Example 1

Create error analysis asynchronously:

```
from snorkelai.sdk.develop import ErrorAnalysis
error_analysis = ErrorAnalysis.create(
    prompt_execution_uid=456,
    sync=False
)
```

Example 2

Create error analysis synchronously:

```
from snorkelai.sdk.develop import ErrorAnalysis
error_analysis = ErrorAnalysis.create(
    prompt_execution_uid=456,
    sync=True
)
```

delete

classmethod **delete**(*error_analysis_run_uid*)

Deletes this error analysis run and all associated cluster data.

Parameters

Name	Type	Default	Info
error_analysis_run_uid	`int`		The unique identifier of the error analysis run to delete.

Raises

ValueError – If the error analysis run has already been deleted or does not exist.

Return type

`None`

Example

```
from snorkelai.sdk.develop import ErrorAnalysis
ErrorAnalysis.delete(error_analysis_run_uid=42)
```

get

classmethod **get**(*error_analysis_run_uid*)

Retrieves an existing error analysis by its unique run identifier.

Parameters

Name	Type	Default	Info
error_analysis_run_uid	`int`		The unique identifier of the error analysis run to retrieve.

Returns

The ErrorAnalysis instance for the specified run, ready for status checks and results retrieval.

Return type

ErrorAnalysis

Raises

ValueError – If no error analysis run exists with the given ID.

Example

```
from snorkelai.sdk.develop import ErrorAnalysis
error_analysis = ErrorAnalysis.get(error_analysis_run_uid=42)
```

get_clusters

get_clusters()

Fetches clusters from a completed error analysis.

Returns

List of cluster objects.

Return type

List[Cluster]

Raises

- **RuntimeError** – If called before analysis is complete.
- **ValueError** – If analysis failed or was deleted.

Example

```python
from snorkelai.sdk.develop import ErrorAnalysis
analysis = ErrorAnalysis.get(error_analysis_run_uid=42)
clusters = analysis.get_clusters()
```

get_latest

classmethod **get_latest**(*prompt_execution_uid*)

Gets the most recent error analysis run for a specific prompt execution.

Parameters

Name	Type	Default	Info
prompt_execution_uid	`int`		The unique identifier of the prompt execution.

Returns

The most recent ErrorAnalysis for the prompt execution or None if no error analysis has been run for this prompt execution.

Return type

ErrorAnalysis or None

Raises

ValueError – If prompt execution does not exist.

Example

```python
from snorkelai.sdk.develop import ErrorAnalysis
latest_analysis = ErrorAnalysis.get_latest(
    prompt_execution_uid=456
)
```

update

update()

Update an error analysis run

Return type

`None`

property **uid***: int*

Return the UID of the error analysis run

snorkelai.sdk.develop.Evaluator

class snorkelai.sdk.develop.Evaluator(*benchmark_uid, criteria_uid, evaluator_uid*)

Bases: `ABC`, `Base`

Base class for all evaluators.

An evaluator assesses a datapoint containing an AI application's response against a specific criteria. Evaluators can be of two types:

- CodeEvaluator: Code-based (using custom Python functions)
- PromptEvaluator: Prompt-based (using LLM prompts)

The goal of an evaluator is to categorize the datapoint into one of the criteria's labels, ultimately assigning the integer associated with the label as that datapoint's score. An evaluator can also assign a rationale for its score, which is used to explain the score.

Read more in the Evaluation overview.

Using the `Evaluator` class requires the following import:

```
from snorkelai.sdk.develop import Evaluator
```

__init__

__init__(*benchmark_uid, criteria_uid, evaluator_uid*)

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark that contains the criteria. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.

evaluator_uid	`int`		The unique identifier for this evaluator.

Methods

`__init__`(benchmark_uid, criteria_uid, ...)	
`create`(*args, **kwargs)	Creates a new evaluator for a criteria.
`delete`(evaluator_uid)	Deletion of an evaluator is not implemented.
`execute`(*args, **kwargs)	Runs the evaluator against all datapoints in the specified dataset split.
`get`(evaluator_uid)	Retrieves the evaluator for a given uid.
`get_execution_result`(execution_uid)	Retrieves the evaluation results and scores for a specific execution.
`get_executions`()	Retrieves all executions for this evaluator.
`get_versions`()	Retrieves all version names for this evaluator.
`poll_execution_result`(execution_uid [, sync])	Polls the evaluation job status and retrieves partial results.
`update`(*args, **kwargs)	Updates the evaluator with a new version.

Attributes

`benchmark_uid`	Return the UID of the parent benchmark
`criteria_uid`	Return the UID of the parent criteria
`evaluator_uid`	Return the UID of the evaluator
`uid`	Return the UID of the evaluator

create

abstract classmethod **create**(**args, **kwargs*)

Creates a new evaluator for a criteria.

Parameters

Name	Type	Default	Info
args	`Any`		Parameters specific to the evaluator type.
kwargs	`Any`		Parameters specific to the evaluator type.

Return type

`Evaluator`

delete

classmethod **delete**(*evaluator_uid*)

Deletion of an evaluator is not implemented.

Parameters

Name	Type	Default	Info
evaluator_uid	`int`		The unique identifier of the evaluator.

Return type

`None`

execute

abstract **execute**(**args, **kwargs*)

Runs the evaluator against all datapoints in the specified dataset split.

Parameters

Name	Type	Default	Info
args	`Any`		Parameters specific to the evaluator type.

kwargs	`Any`		Parameters specific to the evaluator type.

Return type

`int`

get

classmethod **get**(*evaluator_uid*)

Retrieves the evaluator for a given uid.

Parameters

Name	Type	Default	Info
evaluator_uid	`int`		The unique identifier for the evaluator.

Returns

The requested evaluator object.

Return type

Evaluator

Example

```
evaluator = Evaluator.get(evaluator_uid=300)
```

get_execution_result

abstract **get_execution_result**(*execution_uid*)

Retrieves the evaluation results and scores for a specific execution.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The unique identifier of the execution you want to get results for.

Return type

`Dict[str, Dict[str, Union[str, int, float, bool]]]`

get_executions

abstract **get_executions**()

Retrieves all executions for this evaluator.

Return type

`List[Dict[str, Any]]`

get_versions

abstract **get_versions**()

Retrieves all version names for this evaluator.

Return type

`List[str]`

poll_execution_result

abstract **poll_execution_result**(*execution_uid*, *sync=False*)

Polls the evaluation job status and retrieves partial results.

Parameters

Name	Type	Default	Info

execution_uid	`int`		The unique identifier of the execution you want to poll for.
sync	`bool`	`False`	Whether to wait for the job to complete.

Return type

`Tuple`[`str`, `Dict`[`str`, `Dict`[`str`, `Union`[`str`, `int`, `float`, `bool`]]]]

update

abstract **update**(**args, **kwargs*)

Updates the evaluator with a new version.

Parameters

Name	Type	Default	Info
args	`Any`		Parameters specific to the evaluator type.
kwargs	`Any`		Parameters specific to the evaluator type.

Return type

`None`

property **benchmark_uid***: int*

Return the UID of the parent benchmark

property **criteria_uid***: int*

Return the UID of the parent criteria

property **evaluator_uid***: int*

Return the UID of the evaluator

property **uid***: int*

Return the UID of the evaluator

snorkelai.sdk.develop.JsonExportConfig

class snorkelai.sdk.develop.JsonExportConfig

Bases: `object`

Benchmark execution JSON export configuration

__init__

__init__()

Methods

`__init__`()	

snorkelai.sdk.develop.LabelSchema

final class snorkelai.sdk.develop.LabelSchema(*name, uid, dataset_uid, label_map, description, is_text_label=False*)

Bases: `Base`

The LabelSchema object represents a label schema in Snorkel Flow. Currently, this interface only represents Dataset-level (not Node-level) label schemas.

__init__

__init__(*name, uid, dataset_uid, label_map, description, is_text_label=False*)

Create a label schema object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods

Parameters

Name	Type	Default	Info
name	`str`		The name of the label schema.
uid	`int`		The UID for the label schema within Snorkel Flow.
dataset_uid	`int`		The UID for the dataset within Snorkel Flow.
label_map	`Dict[str, int]`		The label map of the label schema.
description	`Optional[str]`		The description of the label schema.
is_text_label	`bool`	`False`	Whether the label schema is a text label schema.

Methods

`__init__`(name, uid, dataset_uid, label_map, ...)	Create a label schema object in-memory with necessary properties.
`copy`(name[, description, label_map, ...])	Copy a label schema.
`create`(dataset_uid, name, data_type, ...[, ...])	Create a label schema for a dataset.
`delete`(label_schema)	Delete a label schema by name or UID.
`get`(label_schema)	Retrieve a label schema by name or UID.
`update`()	Update of a label schema is not implemented.

Attributes

`dataset_uid`	The UID for the dataset within Snorkel Flow.
`description`	The description of the label schema.
`is_text_label`	Whether the label schema is a text label schema.
`label_map`	The label map of the label schema.
`name`	The name of the label schema.
`uid`	The UID for the label schema within Snorkel Flow.

copy

copy(*name, description=None, label_map=None, label_descriptions=None, updated_label_schema=None*)

Copy a label schema.

Parameters

Name	Type	Default	Info
name	`str`		The name of the new label schema.

description	`Optional[str]`	`None`	The description of the new label schema.
label_map	`Optional[Dict[str, int]]`	`None`	The label map of the new label schema.
label_descriptions	`Optional[Dict[str, str]]`	`None`	The label descriptions of the new label schema.
updated_label_schema	`Optional[Dict[str, str]]`	`None`	The update mapping to apply to the new label schema. This is a dictionary mapping label names for the current label schema to those for the new label schema. If a label for the current label schema is removed, it is mapped to None. Examples: 1. Rename “old_1” to “new_1” and remove “old_2”: {“old_1”: “new_1”, “old_2”: None} 2. Merge “old_1” and “old_2” to “new_1”: {“old_1”: “new_1”, “old_2”:

			"new_1"} 3. Split "old_1" to "new_1" and "new_2", and keep assets labeled as "old_1" at "new_1": {"old_1": "new_1"} 4. Add "new_3": None (no change to the existing assets).

Returns

The new label schema object

Return type

LabelSchema

create

classmethod **create**(*dataset_uid, name, data_type, task_type, label_map, multi_label=False, description=None, label_column=None, label_descriptions=None, primary_field=None, is_text_label=False, allow_overlapping=None*)

Create a label schema for a dataset.

Typically, Dataset.create_label_schema() is the recommended entrypoint for creating label schemas.

Parameters

Name	Type	Default	Info
dataset_uid	`int`		The UID for the dataset within Snorkel Flow.

name	`str`		The name of the label schema.
data_type	`str`		The data type of the label schema.
task_type	`str`		The task type of the label schema.
label_map	`Union[Dict[str, int], List[str]]`		A dictionary mapping label names to their integer values, or a list of label names.
multi_label	`bool`	`False`	Whether the label schema is a multi-label schema, by default False.
description	`Optional[str]`	`None`	A description of the label schema, by default None.
label_column	`Optional[str]`	`None`	The name of the column that contains the labels, by default None.
label_descriptions	`Optional[Dict[str, str]]`	`None`	A dictionary mapping label names to their descriptions, by default None.
primary_field	`Optional[str]`	`None`	The primary field of the label schema, by default None.

is_text_label	`bool`	`False`	Whether the label schema is a text label schema, by default False.
allow_overlapping	`Optional[bool]`	`None`	Enable overlapping labels at the same token position. Defaults to None. Sequence tagging only.

Returns

The label schema object

Return type

LabelSchema

delete

classmethod **delete**(*label_schema*)

Delete a label schema by name or UID.

Parameters

Name	Type	Default	Info
label_schema	`Union[str, int]`		The name or UID of the label schema.

Return type

`None`

get

classmethod **get**(*label_schema*)

Retrieve a label schema by name or UID.

Parameters

Name	Type	Default	Info
label_schema	`Union[str, int]`		The name or UID of the label schema.

Returns

The label schema object

Return type

LabelSchema

Raises

ValueError – If no label schema is found with the given name or UID

update

update()

Update of a label schema is not implemented.

Return type

`None`

property **dataset_uid**: *int*

The UID for the dataset within Snorkel Flow.

property **description**: *str | None*

The description of the label schema.

property **is_text_label**: *bool*

Whether the label schema is a text label schema.

property **label_map***: Dict[str, int]*

The label map of the label schema.

property **name***: str*

The name of the label schema.

property **uid***: int*

The UID for the label schema within Snorkel Flow.

snorkelai.sdk.develop.PromptEvaluator

final class snorkelai.sdk.develop.PromptEvaluator(*benchmark_uid, criteria_uid, evaluator_uid, prompt_workflow_uid*)

Bases: `Evaluator`

An evaluator that uses LLM prompts to assess model outputs.

This evaluator type is known as an LLM-as-a-judge (LLMAJ). A prompt evaluator uses LLM prompts to evaluate datapoints containing AI application responses, categorizing them into one of a criteria's labels by assigning the corresponding integer score and optional rationale.

Prompt evaluator execution via the SDK is not yet supported. Please use the GUI to run prompt evaluators.

Read more about LLM-as-a-judge prompts.

Using the `PromptEvaluator` class requires the following import:

```
from snorkelai.sdk.develop import PromptEvaluator
```

__init__

__init__(*benchmark_uid, criteria_uid, evaluator_uid, prompt_workflow_uid*)

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark that contains the criteria.
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
evaluator_uid	`int`		The unique identifier for this evaluator.
prompt_workflow_uid	`int`		The unique identifier of the parent prompt workflow.

Methods

`__init__`(benchmark_uid, criteria_uid, ...)	
`create`(criteria_uid[, user_prompt, ...])	Creates a new prompt evaluator for a criteria.
`delete`(evaluator_uid)	Deletion of an evaluator is not implemented.
`execute`(split[, num_rows, version_name, sync])	Executes the prompt evaluator against a dataset split.
`get`(evaluator_uid)	Retrieves a prompt evaluator for a given uid.
`get_execution_result`(execution_uid)	Retrieves the evaluation results for a specific evaluation execution.
`get_executions`()	Retrieves all executions for this prompt evaluator.
`get_versions`()	Gets all version names for a prompt evaluator.
`poll_execution_result`(execution_uid[, sync])	Polls the job status and retrieves partial results.
`update`([version_name, user_prompt, ...])	Creates a new prompt version for a criteria and updates the evaluator to point to the new prompt version.

Attributes

`benchmark_uid`	Return the UID of the parent benchmark
`criteria_uid`	Return the UID of the parent criteria
`evaluator_uid`	Return the UID of the evaluator
`prompt_workflow_uid`	Return the UID of the parent prompt workflow
`uid`	Return the UID of the evaluator

create

classmethod **create**(*criteria_uid, user_prompt=None, system_prompt=None, model_name=None, fm_hyperparameters=None*)

Creates a new prompt evaluator for a criteria.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier criteria that this evalu assesses.
user_prompt	`Optional[str]`	`None`	The user prompt to u evaluator. At least on user_prompt or syste must be provided.
system_prompt	`Optional[str]`	`None`	The system prompt t evaluator. At least on user_prompt or syste must be provided.
model_name	`Optional[str]`	`None`	The model to use for evaluator.
fm_hyperparameters	`Optional[Dict[str, Any]]`	`None`	The hyperparameters the evaluator. These provided directly to t provider. For example, OpenAI the *response_format* hyperparameter. It ca provided in the follov

			```
			PromptEvaluator
			    criteria_ui
			    user_prompt
			prompt",

			system_prompt="
			prompt",
			    model_name=
			mini",
			    fm_hyperpar
			{

			"response_forma
			            "ty
			"json_object",
			        }
			    }
			)
			```
			 Or a more sophisticat example:

			```PromptEvaluator criteria_ui user_prompt prompt", system_prompt=" prompt", model_name= mini", fm_hyperpar { "response_forma "ty "json_schema", "json_schema": "math_reasoning "schema": { "type": "object "properties": { "steps": { "type": "array" "items": { "type": "object "properties": { "explanation": "string"}, "output": {"typ "string"} },```

			```
			"required":
			["explanation",
			"output"],

			"additionalProp
			False

			}

			},

			"final_answer":
			{"type": "strin


			"required": ["s
			"final_answer"]

			"additionalProp
			False


			"strict": True
			            }
			        }
			    }
			)
			```

Returns

A PromptEvaluator object representing the new evaluator.

Return type

PromptEvaluator

Raises

ValueError – If one of user_prompt, system_prompt is not provided. If model_name is not provided.

execute

execute(*split, num_rows=None, version_name=None, sync=False*)

Executes the prompt evaluator against a dataset split.

This method runs the prompt against the specified dataset split. If no version name is specified, it uses the latest version.

Parameters

Name	Type	Default	Info
split	`str`		The dataset split to evaluate on.
num_rows	`Optional[int]`	`None`	The number of rows to evaluate on.
version_name	`Optional[str]`	`None`	The version name to use for the execution. If not provided, it uses the latest version.
sync	`bool`	`False`	Whether to wait for the execution to complete.

Returns

The execution uid.

Return type

`int`

Example

Example 1

Execute the latest prompt version and poll for results:

```
import time

evaluator = PromptEvaluator.get(evaluator_uid=300)
prompt_execution_uid = evaluator.execute(split="test",
num_rows=100)
while True:
    status, results =
evaluator.poll_execution_result(prompt_execution_uid, sync=False)
    print(f"Job status: {status}")
    if status == "completed" or status == "failed":
        break
    if results:
        print(f"Partial results: {results}")
    time.sleep(10)

print(f"Final results: {results}")
```

Example 2

Execute a specific prompt version and wait for results:

```
evaluator = PromptEvaluator.get(evaluator_uid=300)
prompt_execution_uid = evaluator.execute(split="train",
num_rows=20, version_name="v1.0")
status, results =
evaluator.poll_execution_result(prompt_execution_uid, sync=True)
print(f"Status: {status}")
print(f"Results: {results}")
```

get

classmethod **get**(*evaluator_uid*)

Retrieves a prompt evaluator for a given uid.

Parameters

Name	Type	Default	Info
evaluator_uid	`int`		The unique identifier for the evaluator.

Returns

A PromptEvaluator instance.

Return type

PromptEvaluator

Raises

ValueError – If the evaluator with the given uid is not a PromptEvaluator.

get_execution_result

get_execution_result(*execution_uid*)

Retrieves the evaluation results for a specific evaluation execution.

This method reads the evaluation results for the given evaluation execution UID. If the execution is in progress, it will return partial results.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The evaluation execution UID to get results for.

Returns

A dictionary mapping x_uids to their evaluation results. The evaluation results for each x_uid are a dictionary with the following keys: - “score”: The score for the datapoint - “rationale”: The rationale for the score

Return type

`Dict[str, Dict[str, EvaluationScoreType]]`

get_executions

get_executions()

Retrieves all executions for this prompt evaluator.

This method fetches all executions that have been run using this evaluator. Executions are returned in chronological order, with the oldest execution first.

The dictionary contains the following keys: :rtype: `List`[`Dict`[`str`, `Any`]]

- `execution_uid`: The execution UID
- `created_at`: The timestamp when the execution was created
- `prompt_version_name`: The name of the prompt version used for the execution

Example

Example 1

Get all executions for an evaluator:

```
evaluator = PromptEvaluator.get(evaluator_uid=300)
executions = evaluator.get_executions()
for execution in executions:
    print(f"Execution {execution['execution_uid']}:
{execution['created_at']}")
```

get_versions

get_versions()

Gets all version names for a prompt evaluator.

Returns

A list of version names for the prompt evaluator.

Return type

`List[str]`

poll_execution_result

poll_execution_result(*execution_uid, sync=False*)

Polls the job status and retrieves partial results.

This method checks the current status of the evaluation job and returns both the job status and any available results. The current status can be `running`, `completed`, `failed`, `cancelled`, or `unknown`.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The prompt execution UID to poll for.
sync	`bool`	`False`	Whether to wait for the job to complete. If `False`, returns immediately with current status and partial results.

Return type

`Tuple`[`str`, `Dict`[`str`, `Dict`[`str`, `Union`[`str`, `int`, `float`, `bool`]]]]

Example

Example 1

Poll for job status and partial results:

```
evaluator = PromptEvaluator.get(evaluator_uid=300)
prompt_execution_uid = evaluator.execute(split="test",
num_rows=100)
while True:
    status, results =
evaluator.poll_execution_result(prompt_execution_uid, sync=False)
    print(f"Job status: {status}")
    if results:
        print(f"Partial results: {results}")
    if status == "completed" or status == "failed":
        break
print(f"Final results: {results}")
```

update

update(*version_name=None, user_prompt=None, system_prompt=None, model_name=None, fm_hyperparameters=None*)

Creates a new prompt version for a criteria and updates the evaluator to point to the new prompt version.

Parameters

Name	Type	Default	
version_name	`Optional[str]`	`None`	The name for the nev provided, a default na
user_prompt	`Optional[str]`	`None`	The user prompt to u one of user_prompt c provided.
system_prompt	`Optional[str]`	`None`	The system prompt t one of user_prompt c provided.
model_name	`Optional[str]`	`None`	The model to use for

fm_hyperparameters	`Optional[Dict[str, Any]]`	`None`	The hyperparameters are provided directly For example, OpenAI hyperparameter. It ca way:
			`evaluator =`
			`PromptEvaluator`
			`evaluator.updat`
			`    version_nam`
			`    user_prompt`
			`    system_prom`
			`    model_name=`
			`    fm_hyperpar`
			`        "respon`
			`            "ty`
			`        }`
			`    }`
			`)`
			Or a more sophisticat

```
evaluator =
PromptEvaluator
evaluator.updat
    version_nam
    user_prompt
    system_prom
    model_name=
    fm_hyperpar
        "respon
            "ty
            "js

"math_reasoning





"array",


"type": "object

"properties": {

"explanation":

"output": {"typ


"required": ["e
"output"],

"additionalProp



"final_answer":


["steps", "fina

"additionalProp
```

			`}`
			`}`
			`}`
			`)`

Raises

ValueError – If one of user_prompt, system_prompt is not provided. If model_name is not provided.

Return type

`None`

property **prompt_workflow_uid***: int*

Return the UID of the parent prompt workflow

snorkelai.sdk.develop.Slice

final class **snorkelai.sdk.develop.Slice**(*dataset, slice_uid, name, description=None, templates=None, graph=None*)

Bases: `Base`

Represents a slice within a Snorkel dataset for identifying and managing subsets of datapoints.

A slice is a logical subset of datapoints within a dataset that can be created either manually by adding specific datapoints, or programmatically using slicing functions defined through templates and configurations. Slices are essential for data analysis, model evaluation, and targeted data operations within Snorkel workflows.

Key capabilities:

- Manual datapoint management through add/remove operations
- Programmatic datapoint identification using configurable slicing functions

This class provides methods for creating, retrieving, updating, and managing slice membership. Slice objects should not be instantiated directly - use the `create()` or `get()` class methods instead.

Read more in Using data slices.

Using the `Slice` class requires the following import:

```
from snorkelai.sdk.develop import Slice
```

__init__

__init__(*dataset, slice_uid, name, description=None, templates=None, graph=None*)

Create a Slice object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within

			Snorkel.
slice_uid	`int`		The UID for the slice within Snorkel.
name	`str`		The name of the slice.
description	`Optional[str]`	`None`	The description of the slice.
templates	`Optional[List[Dict[str, Any]]]`	`None`	Configuration defining slicing functions for programmatic datapoint identification.
graph	`Optional[List[Any]]`	`None`	A representation of the slicing function's structure.

Methods

`__init__`(dataset, slice_uid, name[, ...])	Create a Slice object in-memory with necessary properties.
`add_x_uids`(x_uids)	Adds datapoints to a slice.
`create`(dataset, name[, description, ...])	Creates a slice for a dataset.
`delete`(dataset, slice_uid)	Deletes a slice from a dataset.
`get`(dataset, slice)	Retrieves a slice by UID or name.
`get_x_uids`()	Retrieves the UIDs of the datapoints in the slice.
`list`(dataset)	Retrieves all slices for a dataset.
`remove_x_uids`(x_uids)	Removes datapoints from a slice.

`update`([name, description, templates, graph])	Updates the slice properties.

Attributes

`dataset_uid`	Return the UID of the dataset that the slice belongs to
`description`	Return the description of the slice
`name`	Return the name of the slice
`slice_uid`	Return the UID of the slice
`uid`	Return the UID of the slice

add_x_uids

add_x_uids(*x_uids*)

Adds datapoints to a slice.

Parameters

Name	Type	Default	Info
x_uids	`List[str]`		List of UIDs of the datapoints you want to add to the slice.

Raises

Exception – If other server errors occur during the operation.

Return type

`None`

Example

```python
from snorkelai.sdk.develop import Slice
slice = Slice.get(dataset=1, slice=20)
slice.add_x_uids(["uid1", "uid2", "uid3"])
```

create

classmethod **create**(*dataset, name, description='', templates=None, graph=None*)

Creates a slice for a dataset.

Slices are used to identify a subset of datapoints in a dataset. You can add datapoints to a slice manually, or if you define a config, you can add datapoints programmatically. Slice membership can contain both manual and programmatic identified datapoints.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel.
name	`str`		The name of the slice.
description	`str`	`''`	A description of the slice, by default the empty string.
templates	`Optional[List[Dict[str, Any]]]`	`None`	A list of template dictionaries, by default None, you can reference the schema in the `template` module for constructing these templates. These templates are used to

			define the Slicing Function for the slice, allowing it to programmatically add datapoints to the slice membership.
graph	`Optional[List[Any]]`	`None`	A representation of the slicing function's structure, by default None.

Returns

The slice object.

Return type

Slice

Raises

- **ValueError** – If the dataset doesn't exist or cannot be found by name/UID.
- **ValueError** – If the slice name is a reserved name or already exists for the dataset.
- **ValueError** – If there are other validation or server errors during slice creation.

Examples

Example 1

Create a simple slice without templates:

```python
from snorkelai.sdk.develop import Slice
slice = Slice.create(
    dataset=1,
    name="my_slice",
    description="A slice for testing purposes"
)
```

Example 2

Create a slice from a keyword template. This example creates a slice containing all datapoints with the string `capital` in the `Instruction` field:

```python
from snorkelai.sdk.develop import Slice
from snorkelai.sdk.utils.graph import DEFAULT_GRAPH
from snorkelai.templates.keyword_template import
KeywordTemplateSchema

keyword_template = KeywordTemplateSchema(
    operator="CONTAINS",
    keywords=["capital"],
    field="Instruction",
    case_sensitive=False,
    tokenize=True,
)
template_config_dict = {
    **keyword_template.to_dict(),
    "template_type": "keyword"
}
slice = Slice.create(
    dataset=1,
    name="slice_name",
    description="description",
    templates=[
        {
            "transform_type": "dataset_template_filter",
            "config": {
                "transform_config_type": "template_filter_schema",
                "filter_type": "text_template",
                "filter_config_type": "dataset_text_template",
                "dataset_uid": 1,
                "template_config": template_config_dict,
             },
        },
        ],
    graph=DEFAULT_GRAPH,
)
```

delete

classmethod **delete**(*dataset, slice_uid*)

Deletes a slice from a dataset.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel.
slice_uid	`int`		The UID of the slice to delete.

Raises

- **ValueError** – If the dataset or slice doesn't exist or cannot be found by UID.
- **ValueError** – If there are other validation or server errors during slice deletion.

Return type

`None`

Example

```
from snorkelai.sdk.develop import Slice
Slice.delete(dataset=1, slice_uid=20)
```

get

classmethod **get**(*dataset, slice*)

Retrieves a slice by UID or name.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel.
slice	`Union[str, int]`		The UID or name of the slice.

Returns

The slice object.

Return type

Slice

Raises

ValueError – If no slice is found with the given UID or name.

Example

```
from snorkelai.sdk.develop import Slice
slice = Slice.get(dataset=1, slice=20)
```

get_x_uids

get_x_uids()

Retrieves the UIDs of the datapoints in the slice.

Returns

List of UIDs of the datapoints in the slice.

Return type

`List[str]`

Raises

Exception – If other server errors occur during the operation.

Example

```
from snorkelai.sdk.develop import Slice
slice = Slice.get(dataset=1, slice=20)
x_uids = slice.get_x_uids()
```

list

classmethod **list**(*dataset*)

Retrieves all slices for a dataset.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel.

Returns

A list of all the slices available for that dataset.

Return type

List[Slice]

Raises

ValueError – If no dataset is found with the given id.

Example

```
from snorkelai.sdk.develop import Slice
slices = Slice.list(dataset=1)
```

remove_x_uids

remove_x_uids(*x_uids*)

Removes datapoints from a slice.

Parameters

Name	Type	Default	Info
x_uids	`List[str]`		List of UIDs of the datapoints you want to remove from the slice.

Raises

Exception – If other server errors occur during the operation.

Return type

`None`

Example

```
from snorkelai.sdk.develop import Slice
slice = Slice.get(dataset=1, slice=20)
slice.remove_x_uids(["uid1", "uid2", "uid3"])
```

update

update(*name=None, description=None, templates=None, graph=None*)

Updates the slice properties.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new name for the slice, by default None.
description	`Optional[str]`	`None`	The new description for the slice, by default None.
templates	`Optional[List[Dict[str, Any]]]`	`None`	A list of template dictionaries for the slice, by default None.
graph	`Optional[List[str]]`	`None`	A list of strings representing the graph structure for the slice, by default None.

Raises

ValueError – If there are other errors during slice update.

Return type

`None`

Example

```
from snorkelai.sdk.develop import Slice
slice = Slice.get(dataset=1, slice=20)
slice.update(name="new_name", description="updated description")
```

property **dataset_uid***: int*

Return the UID of the dataset that the slice belongs to

property **description***: str | None*

Return the description of the slice

property **name***: str*

Return the name of the slice

property **slice_uid***: int*

Return the UID of the slice

property **uid***: int*

Return the UID of the slice

snorkelai.sdk.develop.User

final class **snorkelai.sdk.develop.User**(*uid, username, role, default_view, email=None, timezone=None, is_superadmin=False, is_active=True*)

Bases: `Base`

User management class for Snorkel SDK.

This class provides methods to create, retrieve, update, and delete users within Snorkel Flow. Each user is assigned a role that determines their permissions, and users are organized within workspaces.

__init__

__init__(*uid, username, role, default_view, email=None, timezone=None, is_superadmin=False, is_active=True*)

Create a user object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()`, `get()`, and `list()` methods

Parameters

Name	Type	Default	Info
uid	`int`		The unique integer identifier for the user within Snorkel Flow.
username	`str`		The username of the user.
role	`UserRole`		The role assigned to the user. Defaults to UserRole.standard if not set.
default_view	`UserView`		The default view preference for the user. Defaults to UserView.standard if not set.
email	`Optional[str]`	`None`	The email address of the user.

timezone	`Optional[str]`	`None`	The timezone preference for the user.
is_superadmin	`bool`	`False`	Whether the user has superadmin privileges. Defaults to False if not set.
is_active	`bool`	`True`	Whether the user account is active. Defaults to True if not set.

Methods

`__init__`(uid, username, role, default_view)	Create a user object in-memory with necessary properties.
`create`(username, password[, role, email, ...])	Create a new user.
`delete`(user)	Delete a user by UID or username.
`get`(user[, workspace])	Get a user by UID or username.
`list`([include_inactive, ...])	List all users in a workspace with optional filters.
`reset_password`(user, new_password)	Reset any user's password without requiring old password.
`update`([email, role, timezone, ...])	Update user properties.

Attributes

`default_view`	The default view preference for the user
`email`	The email address of the user
`is_active`	Whether the user account is active
`is_superadmin`	Whether the user has superadmin privileges
`role`	The role assigned to the user

`timezone`	The timezone preference for the user.
`uid`	The unique integer identifier for the user within Snorkel Flow
`username`	The username of the user

create

classmethod **create**(*username, password, role=None, email=None, timezone=None*)

Create a new user.

Examples

```
>>> from snorkelai.sdk.develop import User, UserRole
>>> new_user = User.create(username='jane_smith',
password='SecurePass123!', role=UserRole.REVIEWER)
User jane_smith has been created with UID 42.
>>> new_user.username
'jane_smith'
```

Parameters

Name	Type	Default	Info
username	`str`		Username for the new user.
password	`str`		Password for the new user (minimum 12 characters with at least 3 character types).
role	`Optional[UserRole]`	`None`	User role - ‘standard’, ‘admin’, ‘reviewer’, ‘labeler’, or ‘superadmin’. Defaults to ‘standard’.
email	`Optional[str]`	`None`	Email address for the user.
timezone	`Optional[str]`	`None`	IANA timezone (e.g., ‘America/Sao_Paulo’, ‘UTC’,

			'Europe/London').

Returns

The created User object

Return type

User

delete

classmethod **delete**(*user*)

Delete a user by UID or username.

This performs a soft delete: sets is_active=False and clears email. Use list(include_inactive=True) to view deleted users.

Examples

```
>>> from snorkelai.sdk.develop import User
>>> User.delete("john_doe")
User john_doe has been deleted.

>>> User.delete(42)
User 42 has been deleted.
```

Parameters

Name	Type	Default	Info
user	`Union[str, int]`		User UID (int) or username (str).

Raises

ValueError – If the user is not found, or if there is a network error or HTTP error during the request.

Return type

`None`

get

classmethod **get**(*user, workspace=None*)

Get a user by UID or username.

Examples

```
>>> from snorkelai.sdk.develop import User
>>> user = User.get("john_doe")
>>> user.username
'john_doe'
>>> user.uid
42
>>> user.email
'john.doe@example.com'

>>> user = User.get(42, workspace="my-workspace")
>>> user.username
'jane_smith'
```

Parameters

Name	Type	Default	Info
user	`Union[str, int]`		User UID (int) or username (str).
workspace	`Union[str, int, None]`	`None`	Workspace UID (int) or workspace name (str). If None, uses the default workspace.

Returns

The user object with all user information

Return type

User

Raises

ValueError – If the user is not found in the specified workspace, or if there is a network error or HTTP error during the request.

list

classmethod **list**(*include_inactive=False, include_superadmins=False, workspace=None*)

List all users in a workspace with optional filters.

Examples

```
>>> from snorkelai.sdk.develop import User
>>> users = User.list()
>>> users[0].username
'john_doe'

>>> inactive_users = User.list(include_inactive=True)
>>> all_users = User.list(include_superadmins=True, workspace="my-
workspace")
```

Parameters

Name	Type	Default	Info
include_inactive	`bool`	`False`	Include inactive/deleted users in the results.
include_superadmins	`bool`	`False`	Include superadmin users in the results.
workspace	`Union[str, int, None]`	`None`	Workspace UID (int) or workspace name (str). If None, uses the default workspace.

Returns

List of User objects matching the filter criteria

Return type

List[User]

reset_password

classmethod **reset_password**(*user, new_password*)

Reset any user's password without requiring old password.

Accepts user UID (int) or username (str).

Examples

```
>>> from snorkelai.sdk.develop import User
>>> User.reset_password("john_doe", "NewSecurePass123!")
Password for user john_doe has been reset.

>>> User.reset_password(42, "AnotherPass456!")
Password for user 42 has been reset.
```

Parameters

Name	Type	Default	Info
user	`Union[str, int]`		User UID (int) or username (str).
new_password	`str`		New password to set for the user.

Raises

ValueError – If the user is not found, or if there is a network error or HTTP error during the request.

Return type

`None`

update

update(*email=None, role=None, timezone=None, is_superadmin=None, default_view=None*)

Update user properties.

Examples

```
>>> from snorkelai.sdk.develop import User
>>> user = User.get("john_doe")
>>> user.update(email="john.doe@example.com")
User john_doe (UID: 42) has been updated.
>>> user.email
'john.doe@example.com'

>>> user.update(timezone="America/New_York")
User john_doe (UID: 42) has been updated.
>>> user.timezone
'America/New_York'
```

Parameters

Name	Type	Default	Info
email	`Optional[str]`	`None`	Email address for the user.
role	`Optional[UserRole]`	`None`	User role - ‘standard’, ‘admin’, ‘reviewer’, ‘labeler’, or ‘superadmin’.
timezone	`Optional[str]`	`None`	IANA timezone (e.g., ‘America/Sao_Paulo’, ‘UTC’, ‘Europe/London’).
is_superadmin	`Optional[bool]`	`None`	Whether the user should be a superadmin.

default_view	`Optional[UserView]`	`None`	Default view for the user.

Return type

`None`

ⓘ NOTE

To deactivate a user (set is_active=False), use User.delete() which performs a soft delete.

property **default_view***: UserView*

The default view preference for the user

property **email***: str | None*

The email address of the user

property **is_active***: bool*

Whether the user account is active

property **is_superadmin***: bool*

Whether the user has superadmin privileges

property **role***: UserRole*

The role assigned to the user

property **timezone***: str | None*

The timezone preference for the user.

property **uid***: int*

The unique integer identifier for the user within Snorkel Flow

property **username***: str*

The username of the user

snorkelai.sdk.develop.UserRole

class snorkelai.sdk.develop.UserRole(*value, names=<not given>, *values, module=None, qualname=None, type=None, start=1, boundary=None*)

Bases: `StrEnum`

__init__

__init__(*args, **kwds)

Methods

`encode`([encoding, errors])	Encode the string using the codec registered for encoding.
`replace`(old, new[, count])	Return a copy with all occurrences of substring old replaced by new.
`split`([sep, maxsplit])	Return a list of the substrings in the string, using sep as the separator string.
`rsplit`([sep, maxsplit])	Return a list of the substrings in the string, using sep as the separator string.
`join`(iterable, /)	Concatenate any number of strings.
`capitalize`()	Return a capitalized version of the string.
`casefold`()	Return a version of the string suitable for caseless comparisons.
`title`()	Return a version of the string where each word is titlecased.
`center`(width[, fillchar])	Return a centered string of length width.
`count`(sub[, start[, end]])	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
`expandtabs`([tabsize])	Return a copy where all tab characters are expanded using spaces.
`find`(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
`partition`(sep, /)	Partition the string into three parts using the given separator.

`index`(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
`ljust`(width[, fillchar])	Return a left-justified string of length width.
`lower`()	Return a copy of the string converted to lowercase.
`lstrip`([chars])	Return a copy of the string with leading whitespace removed.
`rfind`(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
`rindex`(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
`rjust`(width[, fillchar])	Return a right-justified string of length width.
`rstrip`([chars])	Return a copy of the string with trailing whitespace removed.
`rpartition`(sep, /)	Partition the string into three parts using the given separator.
`splitlines`([keepends])	Return a list of the lines in the string, breaking at line boundaries.
`strip`([chars])	Return a copy of the string with leading and trailing whitespace removed.
`swapcase`()	Convert uppercase characters to lowercase and lowercase characters to uppercase.
`translate`(table, /)	Replace each character in the string using the given translation table.
`upper`()	Return a copy of the string converted to uppercase.
`startswith`(prefix[, start[, end]])	Return True if S starts with the specified prefix, False otherwise.
`endswith`(suffix[, start[, end]])	Return True if S ends with the specified suffix, False otherwise.
`removeprefix`(prefix, /)	Return a str with the given prefix string removed if present.
`removesuffix`(suffix, /)	Return a str with the given suffix string removed if present.

`isascii`()	Return True if all characters in the string are ASCII, False otherwise.
`islower`()	Return True if the string is a lowercase string, False otherwise.
`isupper`()	Return True if the string is an uppercase string, False otherwise.
`istitle`()	Return True if the string is a title-cased string, False otherwise.
`isspace`()	Return True if the string is a whitespace string, False otherwise.
`isdecimal`()	Return True if the string is a decimal string, False otherwise.
`isdigit`()	Return True if the string is a digit string, False otherwise.
`isnumeric`()	Return True if the string is a numeric string, False otherwise.
`isalpha`()	Return True if the string is an alphabetic string, False otherwise.
`isalnum`()	Return True if the string is an alpha-numeric string, False otherwise.
`isidentifier`()	Return True if the string is a valid Python identifier, False otherwise.
`isprintable`()	Return True if the string is printable, False otherwise.
`zfill`(width, /)	Pad a numeric string with zeros on the left, to fill a field of the given width.
`format`(*args, **kwargs)	Return a formatted version of S, using substitutions from args and kwargs.
`format_map`(mapping)	Return a formatted version of S, using substitutions from mapping.
`maketrans`	Return a translation table usable for str.translate().
`__init__`(*args, **kwds)	

Attributes

`ADMIN`	

`LABELER`	
`REVIEWER`	
`STANDARD`	
`SUPERADMIN`	

ADMIN *= 'admin'*

LABELER *= 'labeler'*

REVIEWER *= 'reviewer'*

STANDARD *= 'standard'*

SUPERADMIN *= 'superadmin'*

snorkelai.sdk.develop.UserView

class snorkelai.sdk.develop.UserView(*value, names=<not given>, *values, module=None, qualname=None, type=None, start=1, boundary=None*)

Bases: `StrEnum`

__init__

__init__(**args, **kwds*)

Methods

`encode`([encoding, errors])	Encode the string using the codec registered for encoding.
`replace`(old, new[, count])	Return a copy with all occurrences of substring old replaced by new.
`split`([sep, maxsplit])	Return a list of the substrings in the string, using sep as the separator string.
`rsplit`([sep, maxsplit])	Return a list of the substrings in the string, using sep as the separator string.
`join`(iterable, /)	Concatenate any number of strings.
`capitalize`()	Return a capitalized version of the string.
`casefold`()	Return a version of the string suitable for caseless comparisons.
`title`()	Return a version of the string where each word is titlecased.
`center`(width[, fillchar])	Return a centered string of length width.
`count`(sub[, start[, end]])	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
`expandtabs`([tabsize])	Return a copy where all tab characters are expanded using spaces.
`find`(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
`partition`(sep, /)	Partition the string into three parts using the given separator.

`index`(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
`ljust`(width[, fillchar])	Return a left-justified string of length width.
`lower`()	Return a copy of the string converted to lowercase.
`lstrip`([chars])	Return a copy of the string with leading whitespace removed.
`rfind`(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
`rindex`(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
`rjust`(width[, fillchar])	Return a right-justified string of length width.
`rstrip`([chars])	Return a copy of the string with trailing whitespace removed.
`rpartition`(sep, /)	Partition the string into three parts using the given separator.
`splitlines`([keepends])	Return a list of the lines in the string, breaking at line boundaries.
`strip`([chars])	Return a copy of the string with leading and trailing whitespace removed.
`swapcase`()	Convert uppercase characters to lowercase and lowercase characters to uppercase.
`translate`(table, /)	Replace each character in the string using the given translation table.
`upper`()	Return a copy of the string converted to uppercase.
`startswith`(prefix[, start[, end]])	Return True if S starts with the specified prefix, False otherwise.
`endswith`(suffix[, start[, end]])	Return True if S ends with the specified suffix, False otherwise.
`removeprefix`(prefix, /)	Return a str with the given prefix string removed if present.
`removesuffix`(suffix, /)	Return a str with the given suffix string removed if present.

`isascii`()	Return True if all characters in the string are ASCII, False otherwise.
`islower`()	Return True if the string is a lowercase string, False otherwise.
`isupper`()	Return True if the string is an uppercase string, False otherwise.
`istitle`()	Return True if the string is a title-cased string, False otherwise.
`isspace`()	Return True if the string is a whitespace string, False otherwise.
`isdecimal`()	Return True if the string is a decimal string, False otherwise.
`isdigit`()	Return True if the string is a digit string, False otherwise.
`isnumeric`()	Return True if the string is a numeric string, False otherwise.
`isalpha`()	Return True if the string is an alphabetic string, False otherwise.
`isalnum`()	Return True if the string is an alpha-numeric string, False otherwise.
`isidentifier`()	Return True if the string is a valid Python identifier, False otherwise.
`isprintable`()	Return True if the string is printable, False otherwise.
`zfill`(width, /)	Pad a numeric string with zeros on the left, to fill a field of the given width.
`format`(*args, **kwargs)	Return a formatted version of S, using substitutions from args and kwargs.
`format_map`(mapping)	Return a formatted version of S, using substitutions from mapping.
`maketrans`	Return a translation table usable for str.translate().
`__init__`(*args, **kwds)	

Attributes

`ANNOTATION`	

`STANDARD`	

ANNOTATION *= 'annotation'*

STANDARD *= 'standard'*

snorkelai.sdk.utils

Other general utilities.

Functions

`open_file`(path[, mode])	Opens a file at the specified path and returns the corresponding file object for user files.
`resolve_data_path`(path)	Resolve a MinIO path to a local file path.

snorkelai.sdk.utils.open_file

snorkelai.sdk.utils.open_file(*path, mode='r', **kwargs*)

Opens a file at the specified path and returns the corresponding file object for user files.

Currently supports MinIO URLs in the following format: `minio://{bucket}/{key}`, local file paths, and S3 URLs.

New MinIO paths paths are resolved to the current workspace by default, ensuring seamless compatibility. Workspace-scoped paths (e.g., *minio://workspace-1/file.txt*) are also supported if explicitly provided and match the current workspace.

Existing non-workspace-scoped MinIO paths paths will continue to work as expected. However, if a file with the same name is created in a workspace-scoped path, it will take priority moving forward.

Examples

```
>>> # Open a file in MinIO
>>> f = open_file("minio://my-bucket/my-key")
```

```
>>> # Open a file in MinIO outside of the in-platform Notebook
>>> f = open_file(
        "minio://my-bucket/my-key",
        key={MINIO-ACCESS-KEY},
        secret={MINIO-SECRET-KEY},
        client_kwargs={"endpoint_url": {MINIO-URL}}
    )
```

```
>>> # Open a local file
>>> f = open_file("./path/to/file")
```

```
>>> # Open a file in S3
>>> f = open_file("s3://my-bucket/my-key", key={YOUR-S3-ACCESS-KEY},
secret={YOUR-S3-ACCESS-KEY})
```

Parameters

Name	Type	Default	Info
path	`str`		The path of the file to be opened.

mode	`str`	`'r'`	`'w'`, `'rb'`, etc.
kwargs	`Optional[Dict[str, Any]]`		Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc. These options are passed directly to `fsspec.open`.

Returns

A file object opened in the specified mode

Return type

`OpenFile`

snorkelai.sdk.utils.resolve_data_path

snorkelai.sdk.utils.resolve_data_path(*path*)

Resolve a MinIO path to a local file path.

Parameters

Name	Type	Default	Info
path	`str`		MinIO path (e.g., “minio://path/to/file.parquet”).

Returns

File path

Return type

`str`

Examples

```
>>> import pandas as pd
>>> from snorkelai.sdk.utils import resolve_data_path
>>> minio_path = "minio://path/to/file.parquet"
>>> resolved_path = resolve_data_path(minio_path)
>>> df = pd.read_parquet(resolved_path)
```