mlrun

Release UNKNOWN

Iguazio

Feb 18, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

MLRun is a generic and convenient mechanism for data scientists and software developers to describe and run tasks related to machine learning (ML) in various, scalable runtime environments and ML pipelines while automatically tracking executed code, metadata, inputs, and outputs. MLRun integrates with the Nuclio serverless project and with Kubeflow Pipelines.

MLRun features a Python package (`mlrun`), a command-line interface (`mlrun`), and a graphical user interface (the MLRun dashboard).

1.1 Quick-Start

1.1.1 Installation

MLRun requires separate containers for the API and the dashboard (UI).

To install and run MLRun locally using Docker:

```
SHARED_DIR=~/mlrun-data

docker pull mlrun/jupyter:0.5.6
docker pull mlrun/mlrun-ui:0.5.6

docker network create mlrun-network
docker run -it -p 8080:8080 -p 8888:8888 --rm -d --network mlrun-network --name␣
↪jupyter -v ${SHARED_DIR}:/home/jovyan/data mlrun/jupyter:0.5.6
docker run -it -p 4000:80 --rm -d --network mlrun-network --name mlrun-ui -e MLRUN_
↪API_PROXY_URL=http://jupyter:8080 mlrun/mlrun-ui:0.5.6
```

When using Docker MLRun can only use local runs.

To install MLRun on a Kubernetes cluster (e.g., on minikube) and support additional runtimes, run the following commands:

```
kubectl create namespace mlrun
helm repo add stable https://kubernetes-charts.storage.googleapis.com/
helm install -n mlrun nfsprov stable/nfs-server-provisioner
kubectl apply -n mlrun -f https://raw.githubusercontent.com/mlrun/mlrun/master/hack/
↪local/nfs-pvc.yaml
kubectl apply -n mlrun -f https://raw.githubusercontent.com/mlrun/mlrun/master/hack/
↪local/mlrun-local.yaml
kubectl apply -n mlrun -f https://raw.githubusercontent.com/mlrun/mlrun/master/hack/
↪local/mljupy.yaml
```

For more details regarding MLRun installation, refer to the *Installation Guide*.

1.1.2 MLRun Setup

Run the following command from your Python development environment (such as Jupyter Notebook) to install the MLRun package (`mlrun`), which includes a Python API library and the `mlrun` command-line interface (CLI):

```
pip install mlrun
```

Set-up an artifacts path and the MLRun Database path

```
from os import path
from mlrun import mlconf

# Target location for storing pipeline artifacts
artifact_path = path.abspath('jobs')
# MLRun DB path or API service URL
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

print(f'Artifacts path: {artifact_path}\nMLRun DB path: {mlconf.dbpath}')
```

1.1.3 Projects

Projects in the platform are used to package multiple functions, workflows, and artifacts. Projects are created by using the `new_project` MLRun method. Projects are visible in the MLRun dashboard only after they're saved to the MLRun database, which happens whenever you run code for a project.

For example, use the following code to create a project named **my-project** and stores the project definition in a subfolder named `conf`:

```
from mlrun import new_project

project_name = 'my-project'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)

print(f'Project path: {project_path}\nProject name: {project_name}')
```

1.1.4 Experiment Tracking

MLRun introduces the concept of functions, and these functions are part of the project. If you have existing code, the first thing to do is to integrate this code with MLRun. This will not just allow you to run your code in different runtimes, but also enable you to track the function calls, with their inputs and results.

Let's take a simple scenario. First you have some code that reads either a csv file or parquet and returns a DataFrame.

```
import pandas as pd

# Ingest a data set into the platform
def get_data(source_url):

    if source_url.endswith(".csv"):
        df = pd.read_csv(source_url)
    elif source_url.endswith(".parquet") or source_url.endswith(".pq"):
        df = pd.read_parquet(source_url)
    else:
        raise Exception(f"file type unhandled {source_url}")

    return df
```

We would like to do 2 things:

1. Have MLRun handle the data read
2. Log this data to the MLRun database

For this purpose, we'll add a `context` parameter which will be used to log our artifacts. Our code will now look as follows:

```
from os import path
def get_data(context, source_url, format='csv'):

    df = source_url.as_df()

    target_path = path.join(context.artifact_path, 'data')
    # Store the data set in your artifacts database
    context.log_dataset('source_data', df=df, format=format,
                        index=False, artifact_path=target_path)
```

1.1.5 Run Local Code

As input, we will provide a CSV file from S3:

```
# Set the source-data URL
source_url = 'https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv'
```

Next call this function locally, using the `run_local` method. This is a wrapper that will store the execution results in the MLRun database.

```
from mlrun import run_local
get_data_run = run_local(name='get_data',
                         handler=get_data,
                         inputs={'source_url': source_url},
                         project=project_name,
                         artifact_path=artifact_path)
```

When called from python, the output will similar to the following:

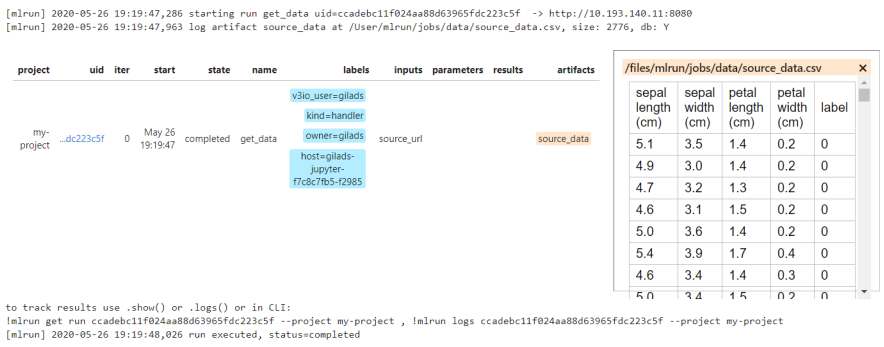
```
[mlrun] 2020-05-26 19:19:47,286 starting run get_data␣
↪uid=b847f1494d45447d9445ea84e1f7271b  -> http://10.193.140.11:8080
[mlrun] 2020-05-26 19:19:47,963 log artifact source_data at /User/mlrun/jobs/data/
↪source_data.csv, size: 2776, db: Y

[mlrun] 2020-05-26 19:19:48,026 run executed, status=completed
```

1.1.6 Experiment Tracking UI

Go to the MLRun UI to see the details of this job. Specifically, for the artifact you will see a preview of the DataFrame data:

As well as statistics:

If you run the function in a Jupyter notebook, the output cell for your function execution will contain a table with run information — including the state of the execution, all inputs and parameters, and the execution results and artifacts. Click on the `source_data` artifact in the **artifacts** column to see a short summary of the data set, as illustrated in the following image:



1.1.7 Functions marketplace

Before implementing your own functions, you should first take a look at the **MLRun functions marketplace** GitHub repository. The marketplace is a centralized location for open-source contributions of function components that are commonly used in machine-learning development.

For example, the `describe` function visualizes the data by creating a histogram, imbalance and correlation matrix plots.

Use the `set_function` MLRun project method, which adds or updates a function object in a project, to load the `describe` marketplace function into a new `describe` project function.

```
project.set_function('hub://describe', 'describe')
```

You can then run the function as part of your project, just as any other function that you have written yourself. To view the function documentation, call the `doc` method:

```
project.func('describe').doc()
```

Which yields the following output:

```
function: describe
describe and visualizes dataset stats
default handler: summarize
entry points:
  summarize: Summarize a table
    context(MLClientCtx)     - the function context
    table(DataItem)          - MLRun input pointing to pandas dataframe (csv/parquet␣
↪file path)
    label_column(str)        - ground truth column label, default=labels
    class_labels(List[str])  - label for each class in tables and plots
    plot_hist(bool)          - (True) set this to False for large tables, default=True
    plots_dest(str)          - destination folder of summary plots (relative to␣
↪artifact_path), default=plots
```

1.1.8 Running functions on different runtimes

One of the key advantages of MLRun is the ability to seamlessly run your code on different runtimes. MLRun can run your code locally, jobs on Kubernetes, on Dask or MPI (Horovod). In this quick-start guide we'll show how to run on Kubernetes.

Accessing shared storage on Kubernetes

The functions need shared storage (file or object) media to pass and store artifacts.

If your are using Iguazio data science platform use the `mount_v3io()` auto-mount modifier. If you use other k8s PVC volumes you can use the `mlrun.platforms.mount_pvc(..)` modifier with the required params.

```
from os import environ
from pathlib import Path
from mlrun.platforms import mount_v3io, mount_pvc

# mount_v3io if using Iguazio data science platform, else mount to k8s PVC 'data'␣
↪directory in the home folder
```

(continues on next page)

(continued from previous page)

```
extra_volume = mount_v3io() if 'V3IO_HOME' in environ \
                            else mount_pvc(pvc_name='nfsvol', volume_name='nfsvol',␣
↪volume_mount_path=path.join(Path.home(), 'data'))
```

Run the function from the marketplace

```
describe = project.func('describe').apply(extra_volume)

describe_run = describe.run(params={'label_column': 'label'},
                            inputs={"table": get_data_run.outputs['source_data']},
                            artifact_path=artifact_path)
```

The expected output is:

```
[mlrun] 2020-05-26 19:20:10,027 starting run describe-summarize␣
↪uid=0655346d86fe48b5bb8c6f34e08873ab  -> http://10.193.140.11:8080
[mlrun] 2020-05-26 19:20:10,174 Job is running in the background, pod: describe-
↪summarize-dp8zq
[mlrun] 2020-05-26 19:20:18,690 log artifact histograms at /User/mlrun/jobs/plots/
↪hist.html, size: 282853, db: Y
[mlrun] 2020-05-26 19:20:19,295 log artifact imbalance at /User/mlrun/jobs/plots/
↪imbalance.html, size: 11716, db: Y
[mlrun] 2020-05-26 19:20:19,517 log artifact correlation at /User/mlrun/jobs/plots/
↪corr.html, size: 30642, db: Y

[mlrun] 2020-05-26 19:20:19,608 run executed, status=completed
```

Convert python code to a function

Place the previously defined `get_data` code in a file called `get_data.py`. We can then use MLRun's `code_to_function` to run that python code as an MLRun function on other runtimes:

```
from mlrun import code_to_function
```

```
get_data_func = code_to_function(name='get-data',
                                 handler='get_data',
                                 filename='get_data.py',
                                 kind='job',
                                 image='mlrun/ml-models')
get_data = project.set_function(get_data_func).apply(extra_volume)
```

Similar to before, we can run this function, but this time, as a job

```
get_data_run = get_data.run(inputs={'source_url': source_url},
                            artifact_path=artifact_path)
```

1.1.9 Create a machine-learning training function

To complete our example, see the following code that trains a simple logistic regression model. In many cases this is not needed since you will find many of the required functions already available in the **Functions Marketplace**. However, this simplified function will give you a good starting point should you need to build similar functions on your own.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

from sklearn import metrics
from pickle import dumps
import matplotlib.pyplot as plt

from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem
from mlrun.artifacts import PlotArtifact

def train(context: MLClientCtx,
          dataset: DataItem,
          label_column: str,
          test_size: float = 0.30,
          random_state: int = 1
         ) -> None:

    """Train a logistic regression model.

    This function accepts a raw dataset and the label column. It splits the data to␣
↪train and test
    datasets and trains a logistic regression model based on the training set. It␣
↪then compares
    the models prediction to the test set.

    :param context:              The MLRun function context used for logging artifacts
    :param dataset:              The name of raw data file containing the dataset
    :param label_column:         The column name of the ground-truth labels within␣
↪the dataset
    :param test_size:            Represent the proportion of the dataset to include␣
↪in the test split
    :param random_state:         Controls the shuffling applied to the data before␣
↪applying the split. Pass an int for reproducible output across multiple function␣
↪calls.
    """

   # Get the dataframe
    df = dataset.as_df()

   # Separate the labael column from the dataframe
    labels = df.pop(label_column)

   # split the sample into train and test and calibration sets:
    xtrain, xtest, ytrain, ytest = train_test_split(df, labels, test_size=test_size,␣
↪random_state=random_state)

   # Log the test set
    test_set = pd.concat([xtest, ytest.to_frame()], axis=1)
    context.log_dataset('test-set', df=test_set, format='parquet', local_path='data/
↪test_set.parquet')
```

(continues on next page)

(continued from previous page)

```
    # Train the model
    model = LogisticRegression(C=1e5).fit(df, labels)

    # Log the model
    context.log_model('model', body=dumps(model), model_dir='models', model_file=
↪'model.pkl')

    # Run the model prediction on the test set
    ypred = model.predict(xtest)

    # Log the results of the test prediction
    context.log_result('accuracy', value=float(metrics.accuracy_score(ytest, ypred)))

    # Log the confusion matrix
    cmd = metrics.plot_confusion_matrix(model, xtest, ytest, normalize='all', values_
↪format='.2g', cmap=plt.cm.Blues)
    context.log_artifact(PlotArtifact('confusion-matrix', body=cmd.figure_), local_
↪path='plots/confusion_matrix.html')
```

Copy the code above to `train.py` file and execute the following command to run that code as a job:

```
train_func = code_to_function(name='train',
                              handler='train',
                              filename='train.py',
                              kind='job',
                              image='mlrun/ml-models')
train_func = project.set_function(train_func_gen).apply(extra_volume)
train_func.apply(mount_v3io()).run(inputs={'dataset': get_data_run.output('source_data
↪')},
                                   params={'label_column': 'label'},
                                   artifact_path=artifact_path)
```

The expected output is:

```
[mlrun] 2020-05-26 19:24:57,635 starting run train-train␣
↪uid=8f99f4702b21470d85b40092ff548e74  -> http://10.193.140.11:8080
[mlrun] 2020-05-26 19:24:57,839 Job is running in the background, pod: train-train-
↪glpnj
[mlrun] 2020-05-26 19:25:03,131 log artifact test-set at /User/mlrun/jobs/data/test_
↪set.parquet, size: 5753, db: Y
[mlrun] 2020-05-26 19:25:03,187 log artifact model at /User/mlrun/jobs/models/, size:␣
↪949, db: Y
[mlrun] 2020-05-26 19:25:03,327 log artifact confusion-matrix at /User/mlrun/jobs/
↪plots/confusion_matrix.html, size: 24535, db: Y

[mlrun] 2020-05-26 19:25:03,354 run executed, status=completed
```

1.1.10 Pipelines

You can also **install Kubeflow Pipelines** and use MLRun to create a workflow that runs the functions defined here.

Once you have Kubeflow installed, copy the following code to **workflow.py** in your project directory (we previously set the project directory as the `conf` folder under the current directory):

```
from kfp import dsl
from os import environ
from pathlib import Path
from mlrun.platforms import mount_v3io, mount_pvc

funcs = {}
LABELS = "label"


# Configure function resources and local settings
def init_functions(functions: dict, project=None, secrets=None):
    extra_volume = mount_v3io() if 'V3IO_HOME' in environ \
                            else mount_pvc(pvc_name='nfsvol', volume_name='nfsvol',␣
↪volume_mount_path=path.join(Path.home(), 'data'))
    for f in functions.values():
        f.apply(extra_volume)

# Create a Kubeflow Pipelines pipeline
@dsl.pipeline(
    name="Quick-start",
    description="This is  simple workflow"
)
def kfpipeline(source_url='https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv
↪'):

    # Ingest the data set
    ingest = funcs['get-data'].as_step(
        name="get-data",
        handler='get_data',
        inputs={'source_url': source_url},
        params={'format': 'pq'},
        outputs=['source_data'])

    # Analyze the dataset
    describe = funcs["describe"].as_step(
        name="summary",
        params={"label_column": LABELS},
        inputs={"table": ingest.outputs['source_data']})

    # Train a model
    train = funcs["train"].as_step(
        name="train",
        params={"label_column": LABELS},
        inputs={"dataset": ingest.outputs['source_data']},
        outputs=['model', 'test_set'])
```

And run the following code:

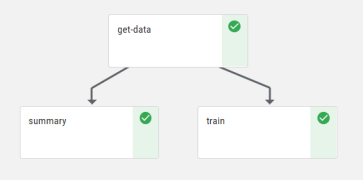
```
# Register the workflow file as "main"
project.set_workflow('main', 'workflow.py')
```

(continues on next page)

(continued from previous page)

```
project.save()

run_id = project.run(
    'main',
    arguments={},
    artifact_path=path.abspath(path.join('pipeline','{{workflow.uid}}')),
    dirty=True)
```

The pipeline created would look as follows



1.2 General Concept and Motivation

1.2.1 The Challenge

As an ML developer or data scientist, you typically want to write code in your preferred local development environment (IDE) or web notebook, and then run the same code on a larger cluster using scale-out containers or functions. When you determine that the code is ready, you or someone else need to transfer the code to an automated ML workflow (for example, using Kubeflow Pipelines). This pipeline should be secure and include capabilities such as logging and monitoring, as well as allow adjustments to relevant components and easy redeployment.

However, the implementation is challenging: various environments (**"runtimes"**) use different configurations, parameters, and data sources. In addition, multiple frameworks and platforms are used to focus on different stages of the development life cycle. This leads to constant development and DevOps/MLOps work.

Furthermore, as your project scales, you need greater computation power or GPUs, and you need to access large-scale data sets. This cannot work on laptops. You need a way to seamlessly run your code on a remote cluster and automatically scale it out.

1.2.2 The MLRun Vision

When ML running experiments, you should ideally be able to record and version your code, configuration, outputs, and associated inputs (lineage), so you can easily reproduce and explain your results. The fact that you probably need to use different types of storage (such as files and AWS S3 buckets) and various databases, further complicates the implementation.

Wouldn't it be great if you could write the code once, using your preferred development environment and simple "local" semantics, and then run it as-is on different platforms? Imagine a layer that automates the build process, execution, data movement, scaling, versioning, parameterization, outputs tracking, and more. A world of easily developed, published, or consumed data or ML "functions" that can be used to form complex and large-scale ML pipelines.

In addition, imagine a marketplace of ML functions that includes both open-source templates and your internally developed functions, to support code reuse across projects and companies and thus further accelerate your work.

This is the goal of MLRun.

> **Note:** The code is in early development stages and is provided as a reference. The hope is to foster wide industry collaboration and make all the resources pluggable, so that developers can code to a single API and use various open-source projects or commercial products.

Back to top

1.2.3 Basic Components

MLRun has the following main components, which are usually grouped into **"projects"**:

- **Project** — a container for all your work on a particular activity. All the associated code, jobs and artifacts are organized within the projects. Projects consist of metadata, source code, workflows, data & artifacts, models, triggers and member management for user collaboration.
- **Function** — a software package with one or more methods and runtime-specific attributes (such as image, command, arguments, and environment). A function can run one or more runs or tasks, it can be created from templates, and it can be stored in a versioned database.
- **Task** — defines the parameters, inputs, and outputs of a logical job or task to execute. A task can be created from a template, and can run over different runtimes or functions.
- **Run** — contains information about an executed task. The run object is created as a result of running a task on a function, and it has all the attributes of a task (such as run parameters and relevant inputs and outputs) with the addition of the execution status and results (including links to output artifacts).
- **Artifact** — versioned data artifacts (such as files, objects, data sets, and models) that are produced or consumed by functions, runs, and workflows.
- **Workflow** — defines a functions pipeline or a directed acyclic graph (DAG) to execute using Kubeflow Pipelines.

Back to top

1.3 Installation Guide

This guide outlines the steps for installing and running MLRun locally.

> **Note:** These instructions use `mlrun` as the namespace (`-n` parameter). You may want to choose a different namespace in your kubernetes cluster.

- *Start Working*

1.3.1 Run MLRun on a Local Docker Registry

To use MLRun with your local Docker registry, run the MLRun API service, dashboard, and example Jupyter server by using the following script.

Note:

- By default the MLRun API service will run inside the Jupyter server, set the MLRUN_DBPATH env var in Jupyter to point to an alternative service address.
- The artifacts and DB will be stored under **/home/jovyan/data**, use docker -v option to persist the content on the host (e.g. `-v ${SHARED_DIR}:/home/jovyan/data`)
- Using Docker is limited to local runtimes.

```
SHARED_DIR=~/mlrun-data

docker pull mlrun/jupyter:0.5.6
docker pull mlrun/mlrun-ui:0.5.6

docker network create mlrun-network
docker run -it -p 8080:8080 -p 8888:8888 --rm -d --network mlrun-network --name␣
↪jupyter -v ${SHARED_DIR}:/home/jovyan/data mlrun/jupyter:0.5.6
docker run -it -p 4000:80 --rm -d --network mlrun-network --name mlrun-ui -e MLRUN_
↪API_PROXY_URL=http://jupyter:8080 mlrun/mlrun-ui:0.5.6
```

When the execution completes —

- Open Jupyter Notebook on port 8888 and run the code in the **examples/mlrun_basics.ipynb** notebook.
- Use the MLRun dashboard on port 4000.

1.3.2 Install MLRun on a Kubernetes Cluster

Perform the following steps to install and run MLRun on a Kubernetes cluster.

Note: The outlined procedure allows using the local, job, and Dask runtimes. To use the MPIJob (Horovod) or Spark runtimes, you need to install additional custom resource definitions (CRDs).

- *Create a namespace*
- *Install a shared volume storage*
- *Install the MLRun API and dashboard (UI) services*

Create a namespace

Create a namespace for mlrun. For example:

```
kubectl create namespace mlrun
```

Install a Shared Volume Storage

You can use any shared file system (or object storage, with some limitations) for sharing artifacts and/or code across containers.

To store data on your Kubernetes cluster itself, you will need to define a **persistent volume**

NFS Server Provisioner

The following example uses a shared NFS server and a Helm chart for the installation:

1. Run the following commands (provided Helm is installed):

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com/
helm install -n mlrun nfsprov stable/nfs-server-provisioner
```

2. Create a **PersistentVolumeClaim** (PVC) for a shared NFS volume by running the following command:

```
kubectl apply -n mlrun -f https://raw.githubusercontent.com/mlrun/mlrun/master/
↪hack/local/nfs-pvc.yaml
```

Install the MLRun API and Dashboard (UI) Services

If you plan to push containers or use a private registry, you need to first create a secret with your Docker registry information. You can do this by running the following command:

```
kubectl create -n mlrun secret docker-registry my-docker --docker-server=https://
↪index.docker.io/v1/ --docker-username=<your-user> --docker-password=<your-password>␣
↪--docker-email=<your-email>
```

Run the following command to apply **mlrun-local.yaml**:

```
kubectl apply -n mlrun -f https://raw.githubusercontent.com/mlrun/mlrun/master/hack/
↪local/mlrun-local.yaml
```

Install a Jupyter Server with a Preloaded MLRun Package.

Run the following command to apply **mljupy.yaml**:

```
kubectl apply -n mlrun -f https://raw.githubusercontent.com/mlrun/mlrun/master/hack/
↪local/mljupy.yaml
```

To change or add packages, see the Jupyter Dockerfile (**Dockerfile.jupy**).

Install Kubeflow

MLRun enables you to run your functions while saving outputs and artifacts in a way that is visible to Kubeflow Pipelines. If you wish to use this capability you will need to install Kubeflow on your cluster. Refer to the **Kubeflow documentation** for more information.

Start Working

- Open Jupyter Notebook on NodePort `30040` and run the code in the **examples/mlrun_basics.ipynb** notebook.
- Use the dashboard at NodePort `30068`.

  Note:

  - You can change the ports by editing the YAML files.
  - You can select to use a Kubernetes Ingress for better security.

1.4 Using MLRun from a Remote Client

This tutorial explains how to use MLRun from a local development environment (IDE) to run jobs on a remote cluster.

1.4.1 In This Document

1.4.2 Prerequisites

Before you begin, ensure that the following prerequisites are met:

1. Install MLRun locally. You can do this by running the following from a command line:

   ```
   pip install mlrun
   ```

2. Ensure that you have remote access to your MLRun service (i.e., to the service's NodePort on the remote Kubernetes cluster).
3. Set environment variables to define your MLRun configuration. As a minimum requirement —
   - Set `MLRUN_DBPATH` to the URL of the remote MLRun database/API service; replace the `<...>` placeholders to identify your remote target:

     ```
     MLRUN_DBPATH=http://<cluster IP>:<port>
     ```

   - If the remote service is on an instance of the Iguazio Data Science Platform (**"the platform"**), set the following environment variables as well; replace the `<...>` placeholders with the information for your specific platform cluster:

```
V3IO_USERNAME=<username of a platform user with MLRun admin privileges>
V3IO_API=<API endpoint of the web-APIs service endpoint; e.g., "webapi.
↪default-tenant.app.mycluster.iguazio.com">
V3IO_ACCESS_KEY=<platform access key>
```

1.4.3 CLI Commands

Use the following commands of the MLRun command-line interface (CLI) — `mlrun` — to build and run MLRun functions:

- *build*
- *run*

The `build` Command

Use the `build` CLI command to build all the function dependencies from the function specification into a function container (Docker image). This command supports many options, including the following; for the full list, run `mlrun build --help`:

```
  --name TEXT             Function name
  --project TEXT          Project name
  --tag TEXT              Function tag
  -i, --image TEXT        Target image path
  -s, --source TEXT       Path/URL of the function source code - a PY file, or a␣
↪directory to archive
                          when using the -a|--archive option (default: './')
  -b, --base-image TEXT   Base Docker image
  -c, --command TEXT      Build commands; for example, '-c pip install pandas'
  --secret-name TEXT      Name of a container-registry secret
  -a, --archive TEXT      Path/URL of a target function-sources archive directory: as␣
↪part of
                          the build, the function sources (see -s|--source) are␣
↪archived into a
                          TAR file and then extracted into the archive directory
  --silent                Don't show build logs
  --with-mlrun            Add the MLRun package ("mlrun")
```

Note: For information about using the `-a|--archive` option to create a function-sources archive, see *Using a Sources Archive* later in this tutorial.

The `run` Command

Use the `run` CLI command to execute a task by using a local or remote function. This command supports many options, including the following; for the full list, run `mlrun run --help`:

```
  -p, --param key=val     Parameter name and value tuples; for example, `-p x=37 -p y=
↪'text'`
  -i, --inputs key=path   Input artifact; for example, `-i infile.txt=s3://mybucket/
↪infile.txt`
  --in-path TEXT          Base directory path/URL for storing input artifacts
  --out-path TEXT         Base directory path/URL for storing output artifacts
  -s, --secrets TEXT      Secrets, either as `file=<filename>` or `env=<ENVAR>,...`;␣
↪for example, `-s file=secrets.txt`
```

(continues on next page)

(continued from previous page)

```
  --name TEXT             Run name
  --project TEXT          Project name or ID
  -f, --func-url TEXT     Path/URL of a YAML function-configuration file, or db://
↪<project>/<name>[:tag] for a DB function object
  --task TEXT             Path/URL of a YAML task-configuration file
  --handler TEXT          Invoke the function handler inside the code file
```

1.4.4 Building and Running a Function from a Git Repository

To build and run a function from a Git repository, start out by adding a YAML function-configuration file in your local environment. This file should describe the function and define its specification. For example, create a **myfunc.yaml** file with the following content in your working directory:

```yaml
kind: job
metadata:
  name: remote-demo1
  project: ''
spec:
  command: 'examples/training.py'
  args: []
  image: .mlrun/func-default-remote-demo-ps-latest
  image_pull_policy: Always
  build:
    #commands: ['pip install pandas']
    base_image: mlrun/mlrun:dev
    source: git://github.com/mlrun/mlrun
```

Then, run the following CLI command and pass the path to your local function-configuration file as an argument to build the function's container image according to the configured requirements. For example, the following command builds the function using the **myfunc.yaml** file from the current directory:

```
mlrun build myfunc.yaml
```

When the build completes, you can use the `run` CLI command to run the function. Set the `-f` option to the path to the local function-configuration file, and pass relevant parameters. For example:

```
mlrun run -f myfunc.yaml -w -p p1=3
```

You can also try the following function-configuration example, which is based on the MLRun CI demo:

```yaml
kind: job
metadata:
  name: remote-git-test
  project: default
  tag: latest
spec:
  command: 'myfunc.py'
  args: []
  image_pull_policy: Always
  build:
    commands: ['pip install pandas']
    base_image: mlrun/mlrun:dev
    source: git://github.com/mlrun/ci-demo.git
```

1.4.5 Using a Sources Archive

The `-a|--archive` option of the CLI *`build`* command enables you to define a remote object path for storing TAR archive files with all the required code dependencies. The remote location can be, for example, in an AWS S3 bucket or in a data container in an Iguazio Data Science Platform ("platform") cluster. Alternatively, you can also set the archive path by using the `MLRUN_DEFAULT_ARCHIVE` environment variable. When an archive path is provided, the remote builder archives the configured function sources (see the `-s|-source` *`build`* option) into a TAR archive file, and then extracts (untars) all of the archive files (i.e., the function sources) into the configured archive location.

To use the archive option, first create a local function-configuration file. For example, you can create a **function.yaml** file in your working directory with the following content; the specification describes the environment to use, defines a Python base image, adds several packages, and defines **examples/training.py** as the application to execute on `run` commands:

```
kind: job
metadata:
  name: remote-demo4
  project: ''
spec:
  command: 'examples/training.py'
  args: []
  image_pull_policy: Always
  build:
    commands: ['pip install mlrun pandas']
    base_image: python:3.6-jessie
```

Next, run the following MLRun CLI command to build the function; replace the `<...>` placeholders to match your configuration:

```
mlrun build <function-configuration file path> -a <archive path/URL> [-s <function-
↪sources path/URL>]
```

Note:

- `.` is a shorthand for a **function.yaml** configuration file in the local working directory.
- The `-a|--archive` option is used to instruct MLRun to create an archive file from the function-code sources at the location specified by the `-s|--sources` option; the default sources location is the current directory (`./`).

For example, the following command uses the **function.yaml** configuration file (`.`), relies on the default function-sources path (`./`), and sets the target archive path to `v3io:///users/$V3IO_USERNAME/tars`. So, for a user named "admin", for example, the function sources from the local working directory will be archived and then extracted into an **admin/tars** directory in the "users" data container of the configured platform cluster (which is accessed via the `v3io` data mount):

```
mlrun build . -a v3io:///users/$V3IO_USERNAME/tars
```

After the function build completes, you can run the function with some parameters. For example:

```
mlrun run -f . -w -p p1=3
```

1.5 End-to-end Pipeline Tutorial

Creating a local function, running predefined functions, creating and running a full ML pipeline with local and library functions.

In this tutorial you will learn how to: - Create and test a simple function - Examine data using serverless (containarized) `describe` function - Create an automated ML pipeline from various library functions - Run and track the pipeline results and artifacts

1.5.1 Create and Test a Local Function (Iris Data Generator)

Import nuclio SDK and magics

```
# nuclio: ignore
import nuclio
```

Specify function dependencies and configuration

```
%%nuclio cmd -c
pip install sklearn
pip install pyarrow
```

```
%nuclio config spec.build.baseImage = "mlrun/mlrun"
```

Function code

Generate the iris dataset and log the dataframe (as csv or parquet file)

```
import os
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.metrics import accuracy_score
from mlrun.artifacts import PlotArtifact
import pandas as pd

def iris_generator(context, format='csv'):
    iris = load_iris()
    iris_dataset = pd.DataFrame(data=iris.data, columns=iris.feature_names)
    iris_labels = pd.DataFrame(data=iris.target, columns=['label'])
    iris_dataset = pd.concat([iris_dataset, iris_labels], axis=1)

    context.logger.info('saving iris dataframe to {}'.format(context.artifact_path))
    context.log_dataset('iris_dataset', df=iris_dataset, format=format, index=False)
```

The following end-code annotation tells `nuclio` to stop parsing the notebook from this cell:

```
# nuclio: end-code
# marks the end of a code section
```

1.5.2 Create a project to host our functions, jobs and artifacts

Projects are used to package multiple functions, workflows, and artifacts. We usually store project code and definitions in a Git archive.

The following code creates a new project in a local dir and initialize git tracking on that

```
from os import path
from mlrun import run_local, mlconf, import_function, mount_v3io
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

# specify artifacts target location
artifact_path = mlconf.artifact_path or path.abspath('./')
project_name = 'sk-project'
```

```
from mlrun import new_project, code_to_function
project_dir = './project'
skproj = new_project(project_name, project_dir, init_git=True)
```

1.5.3 Run the data generator function locally

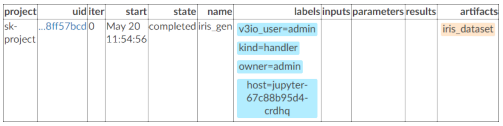
The functions above can be tested locally. Parameters, inputs, and outputs can be specified in the API or the `Task` object. when using `run_local()` the function inputs and outputs are automatically recorded by MLRun experiment and data tracking DB.

In each run we can specify the function, inputs, parameters/hyper-parameters, etc... For more details, see the mlrun_basics notebook.

```
# run the function locally
gen = run_local(name='iris_gen', handler=iris_generator,
                project=project_name, artifact_path=path.join(artifact_path, 'data'))
```

The output would be similar to text below:

```
[mlrun] 2020-05-20 11:54:56,925 starting run iris_gen␣
↪uid=95d9058eac2d48bdb54352e78ff57bcd  -> http://mlrun-api:8080
[mlrun] 2020-05-20 11:54:57,188 saving iris dataframe to /User/artifacts/data
[mlrun] 2020-05-20 11:54:57,268 log artifact iris_dataset at /User/artifacts/data/
↪iris_dataset.csv, size: 2776, db: Y
```



project	uid	iter	start	state	name	labels	inputs	parameters	results	artifacts
sk-project	...8ff57bcd	0	May 20 11:54:56	completed	iris_gen	v3io_user=admin kind=handler owner=admin host=jupyter-67c88b95d4-crdhq				iris_dataset

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 95d9058eac2d48bdb54352e78ff57bcd --project sk-project , !mlrun logs␣
↪95d9058eac2d48bdb54352e78ff57bcd --project sk-project
[mlrun] 2020-05-20 11:54:57,373 run executed, status=completed
```

Convert our local code to a distributed serverless function object

```
gen_func = code_to_function(name='gen_iris', kind='job')
skproj.set_function(gen_func)
```

1.5.4 Load and run a library function (visualize dataset features and stats)

Step 1: load the function object from the function hub (marketplace) > note: the function marketplace location is configurable, by default it points to `mlrun/functions` git

```
skproj.set_function('hub://describe', 'describe')
```

```
# read the remote function doc, params, usage
skproj.func('describe').doc()
#skproj.func('describe').spec.image_pull_policy = 'Always'
```

```
function: describe
describe and visualizes dataset stats
default handler: summarize
entry points:
  summarize: Summarize a table
    context(MLClientCtx)  - the function context
    table(DataItem)  - MLRun input pointing to pandas dataframe (csv/parquet file␣
↪path)
    label_column(str)  - ground truth column label, default=labels
    class_labels(List[str])  - label for each class in tables and plots
    plot_hist(bool)  - (True) set this to False for large tables, default=True
    plots_dest(str)  - destination folder of summary plots (relative to artifact_
↪path), default=plots
```

Step 2: Run the describe function as a Kubernetes job with specified parameters.

> `mount_v3io()` connects our function to v3io shared file system and allow us to pass the data and get back the results (plots) directly to our notebook, we can choose other mount options to use NFS or object storage

```
skproj.func('describe').apply(mount_v3io()).run(params={'label_column': 'label'},
                                                 inputs={"table": gen.outputs['iris_
↪dataset']},
                                                 artifact_path=artifact_path)
```

```
[mlrun] 2020-05-20 11:55:01,994 starting run describe-summarize␣
↪uid=9fc84dd77c4142af995c33244ef870b6  -> http://mlrun-api:8080
[mlrun] 2020-05-20 11:55:02,173 Job is running in the background, pod: describe-
↪summarize-x6r9q
[mlrun] 2020-05-20 11:55:12,627 starting local run: main.py # summarize
[mlrun] 2020-05-20 11:55:16,068 log artifact histograms at /User/artifacts/plots/hist.
↪html, size: 282853, db: Y
[mlrun] 2020-05-20 11:55:16,597 log artifact imbalance at /User/artifacts/plots/
↪imbalance.html, size: 11716, db: Y
[mlrun] 2020-05-20 11:55:16,765 log artifact correlation at /User/artifacts/plots/
↪corr.html, size: 30642, db: Y

[mlrun] 2020-05-20 11:55:16,837 run executed, status=completed
final state: succeeded
```

project	uid	iter	start	state	name	labels	inputs	parameters	results	artifacts
sk-project	...4ef870b6	0	May 20 11:55:13	completed	describe-summarize	host=describe-summarize-x6r9q kind=job owner=admin v3io_user=admin	table	label_column=label	scale_pos_weight=1.00	histograms imbalance correlation

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 9fc84dd77c4142af995c33244ef870b6 --project sk-project , !mlrun logs␣
↪9fc84dd77c4142af995c33244ef870b6 --project sk-project
[mlrun] 2020-05-20 11:55:21,550 run executed, status=completed
```

1.5.5 Create a Fully Automated ML Pipeline

Add more functions to our project to be used in our pipeline (from the functions hub/marketplace)

AutoML training (classifier), Model validation (test_classifier), Real-time model server, and Model REST API Tester

```
skproj.set_function('hub://sklearn_classifier', 'train')
skproj.set_function('hub://test_classifier', 'test')
skproj.set_function('hub://model_server', 'serving')
skproj.set_function('hub://model_server_tester', 'live_tester')
#print(skproj.to_yaml())
```

Define and save a pipeline

The following workflow definition will be written into a file, it describes a Kubeflow execution graph (DAG) and how functions and data are connected to form an end to end pipeline.

- Build the iris generator (ingest) function container
- Ingest the iris data
- Analyze the dataset (describe)
- Train and test the model
- Deploy the model as a real-time serverless function
- Test the serverless function REST API with test dataset

Check the code below to see how functions objects are initialized and used (by name) inside the workflow. The `workflow.py` file has two parts, initialize the function objects and define pipeline dsl (connect the function inputs and outputs).

> Note: the pipeline can include CI steps like building container images and deploying models as illustrated in the following example.

```
%%writefile project/workflow.py
from kfp import dsl
from mlrun import mount_v3io

funcs = {}
DATASET = 'iris_dataset'
LABELS  = "label"


# init functions is used to configure function resources and local settings
def init_functions(functions: dict, project=None, secrets=None):
```

(continues on next page)

(continued from previous page)

```
    for f in functions.values():
        f.apply(mount_v3io())

    # uncomment this line to collect the inference results into a stream
    # and specify a path in V3IO (<datacontainer>/<subpath>)
    #functions['serving'].set_env('INFERENCE_STREAM', 'users/admin/model_stream')


@dsl.pipeline(
    name="Demo training pipeline",
    description="Shows how to use mlrun."
)
def kfpipeline():

    # build our ingestion function (container image)
    builder = funcs['gen-iris'].deploy_step(skip_deployed=True)

    # run the ingestion function with the new image and params
    ingest = funcs['gen-iris'].as_step(
        name="get-data",
        handler='iris_generator',
        image=builder.outputs['image'],
        params={'format': 'pq'},
        outputs=[DATASET])

    # analyze our dataset
    describe = funcs["describe"].as_step(
        name="summary",
        params={"label_column": LABELS},
        inputs={"table": ingest.outputs[DATASET]})

    # train with hyper-paremeters
    train = funcs["train"].as_step(
        name="train-skrf",
        params={"sample"          : -1,
                "label_column"    : LABELS,
                "test_size"       : 0.10},
        hyperparams={'model_pkg_class': ["sklearn.ensemble.RandomForestClassifier",
                                         "sklearn.linear_model.LogisticRegression",
                                         "sklearn.ensemble.AdaBoostClassifier"]},
        selector='max.accuracy',
        inputs={"dataset"         : ingest.outputs[DATASET]},
        outputs=['model', 'test_set'])

    # test and visualize our model
    test = funcs["test"].as_step(
        name="test",
        params={"label_column": LABELS},
        inputs={"models_path" : train.outputs['model'],
                "test_set"    : train.outputs['test_set']})

    # deploy our model as a serverless function
    deploy = funcs["serving"].deploy_step(models={f"{DATASET}_v1": train.outputs[
↪'model']}, tag='v2')

    # test out new model server (via REST API calls)
    tester = funcs["live_tester"].as_step(name='model-tester',
```

(continues on next page)

(continued from previous page)

```
        params={'addr': deploy.outputs['endpoint'], 'model': f"{DATASET}_v1"},
        inputs={'table': train.outputs['test_set']})
```

```
Overwriting project/workflow.py
```

```
# register the workflow file as "main", embed the workflow code into the project YAML
skproj.set_workflow('main', 'workflow.py', embed=True)
```

Save the project definitions to a file (project.yaml), it is recommended to commit all changes to a Git repo.

```
skproj.save()
```

Run a pipeline workflow

Use the `run` method to execute a workflow, you can provide alternative arguments and specify the default target for workflow artifacts. The workflow ID is returned and can be used to track the progress or you can use the hyperlinks

> Note: The same command can be issued through CLI commands: `mlrun project my-proj/ -r main -p "v3io:///users/admin/mlrun/kfp/{{workflow.uid}}/"`

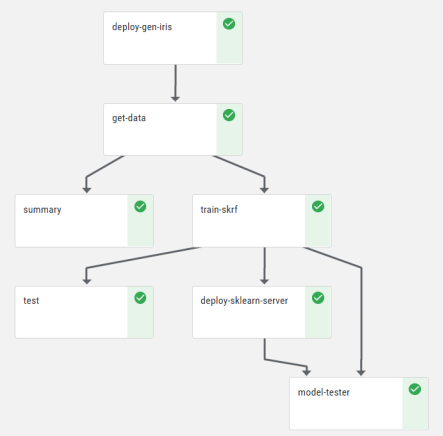
The dirty flag allow us to run a project with uncommited changes (when the notebook is in the same git dir it will always be dirty)

```
artifact_path = path.abspath('./pipe/{{workflow.uid}}')
run_id = skproj.run(
    'main',
    arguments={},
    artifact_path=artifact_path,
    dirty=True)
```

This would output:

```
[mlrun] 2020-05-20 11:55:22,685 Pipeline run id=64d6f1e7-a582-4180-bba6-52c4a860d46b,␣
↪check UI or DB for progress
```

Visually, the workflow would look as follows:



Track pipeline results

```
from mlrun import get_run_db
db = get_run_db().connect()
db.list_runs(project=skproj.name, labels=f'workflow={run_id}').show()
```

project	uid	iter	start	state	name	labels	inputs	parameters	results	artifacts
sk-project	...671c5df2	0	Apr 10 20:51:29	running	train-skrf	kind=job owner=admin v3io_user=admin workflow=cfce7566-0446-400c-bb88-8688d7776c91	dataset	label_column=label sample=-1 test_size=0.1		
sk-project	...ce1fa27f	0	Apr 10 20:51:29	running	summary	kind=job owner=admin v3io_user=admin workflow=cfce7566-0446-400c-bb88-8688d7776c91	table	label_column=label		
sk-project	...4e574ffd	0	Apr 10 20:51:20	completed	get-data	host=get-data-mkrmx kind=job owner=admin v3io_user=admin workflow=cfce7566-0446-400c-bb88-8688d7776c91		format=pq		iris_dataset

back to top

1.6 Data Management and Versioning

1.6.1 Overview

An artifact is any data that is produced and/or consumed by functions or jobs.

The artifacts are stored in the project and are divided to 3 main types:

1. **Datasets** — any data , such as tables and DataFrames.
2. **Plots** — images, figures, and plotlines.
3. **Models** — all trained models.

From the projects page, click on the **Artifacts** link to view all the artifacts stored in the project

You can search the artifacts based on time and labels. In the Monitor view, you can view per artifact its location, the artifact type, labels, the producer of the artifact, the artifact owner, last update date.

Per each artifact you can view its content as well as download the artifact.

1.6.2 Datasets

Storing datasets is important in order to have a record of the data that was used to train the model, as well as storing any processed data. MLRun comes with built-in support for DataFrame format, and can not just store the DataFrame, but also provide the user information regarding the data, such as statistics.

The simplest way to store a dataset is with the following code:

```
context.log_dataset(key='my_data', df=df)
```

Where `key` is the the name of the artifact and `df` is the DataFrame. By default, MLRun will store a short preview of 20 lines. You can change the number of lines by using the `preview` parameter and setting it to a different value.

MLRun will also calculate statistics on the DataFrame on all numeric fields. You can enable statistics regardless to the DataFrame size by setting the `stats` parameter to `True`.

Logging a Dataset From a Job

The following example shows how to work with datasets from a job:

```
from os import path
from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem

# Ingest a data set into the platform
def get_data(context: MLClientCtx, source_url: DataItem, format: str = 'csv'):

    iris_dataset = source_url.as_df()

    target_path = path.join(context.artifact_path, 'data')
    # Optionally print data to your logger
    context.logger.info('Saving Iris data set to {} ...'.format(target_path))

    # Store the data set in your artifacts database
    context.log_dataset('iris_dataset', df=iris_dataset, format=format,
                        index=False, artifact_path=target_path)
```

We can run this function locally or as a job. For example if we run it locally:

```
from os import path
from mlrun import new_project, run_local, mlconf

project_name = 'my-project'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)

# Target location for storing pipeline artifacts
artifact_path = path.abspath('jobs')
# MLRun DB path or API service URL
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

source_url = 'https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv'
# Run get-data function locally
get_data_run = run_local(name='get_data',
                         handler=get_data,
                         inputs={'source_url': source_url},
```

(continues on next page)

(continued from previous page)

```
                          project=project_name,
                          artifact_path=artifact_path)
```

The dataset location is returned in the `outputs` field, therefore you can get the location by calling `get_data_run.outputs['iris_dataset']` and use the `get_dataitem` function to get the dataset itself.

```
# Read your data set
from mlrun.run import get_dataitem
dataset = get_dataitem(get_data_run.outputs['iris_dataset'])
```

Call `dataset.meta.stats` to obtain the data statistics. You can also get the data as a Pandas Dataframe by calling the `dataset.as_df()`.

1.6.3 Models

An essential piece of artifact management and versioning is storing a model version. This allows the users to experiment with different models and compare their performance, without having to worry about losing their previous results.

The simplest way to store a model named `my_model` is with the following code:

```
from pickle import dumps
model_data = dumps(model)
context.log_model(key='my_model', body=model_data, model_file='my_model.pkl')
```

You can also store any related metrics by providing a dictionary in the `metrics` parameter, such as `metrics={'accuracy': 0.9}`. Furthermore, any additional data that you would like to store along with the model can be specified in the `extra_data` parameter. For example `extra_data={'confusion': confusion.target_path}`

A convenient utility method, `eval_model_v2`, which calculates mode metrics is available in `mlrun.utils`.

See example below for a simple model trained using scikit-learn (normally, you would send the data as input to the function). The last 2 lines evaluate the model and log the model.

```
from sklearn import linear_model
from sklearn import datasets
from sklearn.model_selection import train_test_split
from pickle import dumps

from mlrun.execution import MLClientCtx
from mlrun.mlutils import eval_model_v2

def train_iris(context: MLClientCtx):

    # Basic scikit-learn iris SVM model
    X, y = datasets.load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)
    model = linear_model.LogisticRegression(max_iter=10000)
    model.fit(X_train, y_train)

    # Evaluate model results and get the evaluation metrics
    eval_metrics = eval_model_v2(context, X_test, y_test, model)

    # Log model
```

(continues on next page)

(continued from previous page)

```
    context.log_model("model",
                      body=dumps(model),
                      artifact_path=context.artifact_subpath("models"),
                      extra_data=eval_metrics,
                      model_file="model.pkl",
                      metrics=context.results,
                      labels={"class": "sklearn.linear_model.LogisticRegression"})
```

Save the code above to `train_iris.py`. The following code loads the function and runs it as a job. See the quick-start page to learn how to create the project and set the artifact path.

```
from mlrun import code_to_function

gen_func = code_to_function(name=train_iris,
                            filename='train_iris.py',
                            handler=train_iris,
                            kind='job',
                            image='mlrun/ml-models')

train_iris_func = project.set_function(gen_func).apply(auto_mount())

train_iris = train_iris_func.run(name=train_iris,
                                 handler=train_iris,
                                 artifact_path=artifact_path)
```

You can now use `get_model` to read the model and run it. This function will get the model file, metadata, and extra data. The input can be either the path of the model, or the directory where the model resides. If you provide a directory, the function will search for the model file (by default it searches for .pkl files)

The following example gets the model from `models_path` and test data in `test_set` with the expected label provided as a column of the test data. The name of the column containing the expected label is provided in `label_column`. The example then retrieves the models, runs the model with the test data and updates the model with the metrics and results of the test data.

```
from pickle import load

from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem
from mlrun.artifacts import get_model, update_model
from mlrun.mlutils import eval_model_v2

def test_model(context: MLClientCtx,
               models_path: DataItem,
               test_set: DataItem,
               label_column: str):

    if models_path is None:
        models_path = context.artifact_subpath("models")
    xtest = test_set.as_df()
    ytest = xtest.pop(label_column)

    model_file, model_obj, _ = get_model(models_path)
    model = load(open(model_file, 'rb'))

    extra_data = eval_model_v2(context, xtest, ytest.values, model)
    update_model(model_artifact=model_obj, extra_data=extra_data,
                 metrics=context.results, key_prefix='validation-')
```

To run the code, place the code above in `test_model.py` and use the following snippet. The model from the previous step is provided as the `models_path`:

```
from mlrun.platforms import auto_mount
gen_func = code_to_function(name=test_model,
                            filename='test_model.py',
                            handler=test_model,
                            kind='job',
                            image='mlrun/ml-models')

func = project.set_function(gen_func).apply(auto_mount())

run = func.run(name=test_model,
               handler=test_model,
               params={'label_column': 'label'},
               inputs={'models_path': train_iris.outputs['model'],
                       'test_set': 'https://s3.wasabisys.com/iguazio/data/iris/iris_
↪dataset.csv'}),
               artifact_path=artifact_path)
```

1.6.4 Plots

Storing plots is useful to visualize the data and to show any information regarding the model performance. For example, one can store scatter plots, histograms and cross-correlation of the data, and for the model store the ROC curve and confusion matrix.

For example, the following code creates a confusion matrix plot using sklearn.metrics.plot_confusion_matrix and stores the plot in the artifact repository:

```
from mlrun.artifacts import PlotArtifact
from mlrun.mlutils import gcf_clear

gcf_clear(plt)
confusion_matrix = metrics.plot_confusion_matrix(model,
                                                 xtest,
                                                 ytest,
                                                 normalize='all',
                                                 values_format = '.2g',
                                                 cmap=plt.cm.Blues)
confusion_matrix = context.log_artifact(PlotArtifact('confusion-matrix',␣
↪body=confusion_matrix.figure_),
                                        local_path='plots/confusion_matrix.html')
```

You can use the `update_dataset_meta` function to associate the plot with the dataset by assigning the value of the `extra_data` parameter:

```
from mlrun.artifacts import update_dataset_meta

extra_data = {'confusion_matrix': confusion_matrix}
update_dataset_meta(dataset, extra_data=extra_data)
```

Back to top

1.7 Projects

A Project is a container for all your work on a particular activity. All the associated code, jobs and artifacts are organized within the projects. A project is also a great way to collaborate with others, since you can share your work, as well as create projects based on existing projects.

- *Creating a new project*
- *Setting up Git Remote Repository*
- *Loading existing projects*
- *Updating the project and code*

1.7.1 Creating a new project

It's a best practice to have all your notebooks associated with a project. An easy way to do that is to create a project in the beginning of the notebook using the `new_project` MLRun method, which receives the following parameters:

- **`name`** (Required) — the project name.
- **`context`** — the path to a local project directory (the project's context directory). The project directory contains a project-configuration file (default: **project.yaml**), which defines the project, and additional generated Python code. The project file is created when you save your project (using the `save` MLRun project method), as demonstrated in Step 6.
- **`functions`** — a list of functions objects or links to function code or objects.
- **`init_git`** — set to `True` to perform Git initialization of the project directory (`context`).

  Note: It's customary to store project code and definitions in a Git repository.

Projects are visible in the MLRun dashboard only after they're saved to the MLRun database, which happens whenever you run code for a project.

For example, use the following code to create a project named **my-project** and stores the project definition in a subfolder named `conf`:

```
from os import path
from mlrun import new_project

project_name = 'my-project'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)

print(f'Project path: {project_path}\nProject name: {project_name}')
```

You can also ensure the project name is unique, by concatenating your current username. For example, the following code concatenates the **V3IO_USERNAME** environment variable to the project name:

```
from os import getenv
project_name = '-'.join(filter(None, ['my-project', getenv('V3IO_USERNAME', None)]))
```

1.7.2 Setting up Git Remote Repository

It is also highly recommended to set up a remote repository in order to save your work on git. To do that, you need to call `create_remote`. For example, to set up a remote repository on GitHub:

```
remote_git = 'https://github.com/<my-org>/<my-repo>.git'
project.create_remote(remote_git)
```

In case the remote repository has existing content, you should pull from it:

```
project.pull()
```

1.7.3 Loading existing projects

You can use an existing project as a baseline by calling the `load_project` function. This enables reuse of existing code and definitions.

Projects can be stored in a Git repo or in a tar archive (on object store like S3, V3IO).

For example, to load the **demo-xgb-project** to `my_proj` the user's home directory:

```
from os import path
from pathlib import Path
# source Git Repo
# YOU SHOULD fork this to your account and use the fork if you plan on modifying the␣
↪code
url = 'git://github.com/mlrun/demo-xgb-project.git' # refs/tags/v0.4.7'

# alternatively can use tar files, e.g.
#url = 'v3io:///users/admin/tars/src-project.tar.gz'

# change if you want to clone into a different dir, can use clone=True to override␣
↪the dir content
project_dir = path.join(str(Path.home()), 'my_proj')
project = load_project(project_dir, url, clone=True)
```

> **Note**: If URL is not specified it will use the context and search for Git repo inside it, or use the init_git=True flag to initialize a Git repo in the target context directory.

1.7.4 Updating the project and code

A user can update the code using the standard Git process (commit, push, ..), if you update/edit the project object you need to run `proj.save()` which will update the `project.yaml` file in your context directory, followed by pushing your updates.

You can use `proj.push(branch, commit_message, add=[])` which will do the work for you (save the yaml, commit updates, push)

> Note: you must push updates before you build functions or run workflows since the builder will pull the code from the git repo.

If you are using containerized Jupyter you may need to first set your Git parameters, e.g. using the following commands:

```
git config --global user.email "<my@email.com>"
git config --global user.name "<name>"
git config --global credential.helper store
```

After that you would need to login once to git with your password as well as restart the notebook

```
project.push('master', 'some edits')
```

If you want to pull changes done by others use `proj.pull()`, if you need to update the project spec (since the yaml file changed) use `proj.reload()` and for updating the local/remote function specs use `proj.sync_functions()` (or add `sync=True` to `.reload()`).

```
project.pull()
```

Back to top

1.8 Loading MLRun functions from the marketplace

1.8.1 Overview

In this tutorial we'll demonstrate how to import a function from the marketplace into your own project and provide some basic instructions of how to run the function and view their results.

1.8.2 Functions Marketplace

MLRun marketplace has a wide range of functions that can be used for a variety of use cases. In the marketplace there are functions for ETL, data preparation, training (ML & Deep learning), serving, alerts and notifications and etc.. Each function has a docstring that explains how to use it and in addition the functions are associated with categories to make it easier for the user to find the relevant one.

Functions can be easily imported into your project and therefore help users to speed up their development cycle by reusing built-in code.

1.8.3 Searching for functions

The Marketplace is stored in this GitHub repo: https://github.com/mlrun/functions In the README file you can view the list of functions in the marketplace and their categories.

1.8.4 Setting the project configuration

The first step for each project is to set the project name and path:

```
from os import path, getenv
from mlrun import new_project

project_name = 'load-func'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)

print(f'Project path: {project_path}\nProject name: {project_name}')
```

Set the artifacts path

The artifact path is the default path for saving all the artifacts that the functions generate:

```
from mlrun import run_local, mlconf, import_function, mount_v3io

# Target location for storing pipeline artifacts
artifact_path = path.abspath('jobs')
# MLRun DB path or API service URL
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

print(f'Artifacts path: {artifact_path}\nMLRun DB path: {mlconf.dbpath}')
```

1.8.5 Loading function from the marketplace

Loading functions is done by running `project.set_function` `set_function` updates or adds a function object to the project

`set_function(func, name='', kind='', image=None, with_repo=None)`

Parameters:

- **func** – function object or spec/code url
- **name** – name of the function (under the project)
- **kind** – runtime kind e.g. job, nuclio, spark, dask, mpijob default: job
- **image** – docker image to be used, can also be specified in the function object/yaml
- **with_repo** – add (clone) the current repo to the build source

Returns: project object

For more information see the `set_function` API documentation.

Load function Example

In this example we load the describe function. this function analyze a csv or parquet file for data analysis

```
project.set_function('hub://describe', 'describe')
```

Create a function object called my_describe:

```
my_describe = project.func('describe')
```

1.8.6 View the function params

In order to view the parameters run the function with .doc()

```
my_describe.doc()
```

```
    function: describe
    describe and visualizes dataset stats
    default handler: summarize
    entry points:
      summarize: Summarize a table
        context(MLClientCtx)  - the function context, default=
        table(DataItem)  - MLRun input pointing to pandas dataframe (csv/parquet file␣
↪path), default=
        label_column(str)  - ground truth column label, default=None
        class_labels(List[str])  - label for each class in tables and plots,␣
↪default=[]
        plot_hist(bool)  - (True) set this to False for large tables, default=True
        plots_dest(str)  - destination folder of summary plots (relative to artifact_
↪path), default=plots
        update_dataset  - when the table is a registered dataset update the charts in-
↪place, default=False
```

1.8.7 Running the function

Use the `run` method to to run the function.

When working with functions pay attention to the following:

- Input vs params - for sending data items to a function, users should send it via "inputs" and not as params.
- Working with artifacts - Artifacts from each run are stored in the artifact_path which can be set globally through environment variable (MLRUN_ARTIFACT_PATH) or through the config, if its not already set we can create a directory and use it in our runs. Using {{run.uid}} in the path will allow us to create a unique directory per run, when we use pipelines we can use the {{workflow.uid}} template option.

In this example we run the describe function. this function analyze a dataset (in our case it's a csv file) and generate html files (e.g. correlation, histogram) and save them under the artifact path

```
DATA_URL = 'https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv'

my_describe.run(name='describe',
                inputs={'table': DATA_URL},
                artifact_path=artifact_path)
```

Saving the artifacts in a unique folder for each run

```
out = mlconf.artifact_path or path.abspath('./data')
my_describe.run(name='describe',
                inputs={'table': DATA_URL},
                artifact_path=path.join(out, '{{run.uid}}'))
```

Viewing the jobs & the artifacts

There are few options to view the outputs of the jobs we ran:

- In Jupyter - the result of the job is displayed in Jupyter notebook. Note that when you click on the artifacts it displays its content in Jupyter.
- UI - going to the MLRun UI, under the project name, you can view the job that was running as well as the artifacts it was generating

1.9 Job Submission and Tracking

1.9.1 Experiment Tracking

Experiment tracking enables you to store every action and result in your project. It is a convenient way to go back to previous results and compare different artifacts. You will find the following sections within your project:

1. ***Artifacts***: Any data stored is considered an artifact. Artifacts are versioned and enable you to compare different outputs of the executed Jobs.
2. ***Functions***: The code in your project is stored in functions that are versioned. Functions can the functions you wrote, or externally loaded functions, such as functions that originate from the MLRun Functions Marketplace.
3. ***Jobs***: Allows you to review anything you executed, and review the execution outcome.
4. ***Pipelines***: Reusable end-to-end ML workflows.

You can compare different experiments and review these results. When using experiment tracking you don't have to worry about saving your work as you try out different models and various configurations, you can always compare your different results and choose the best strategy based on your current and past experiments.

Experiments are also a great way to collaborate. Your colleagues can review the different steps you took and try out other scenarios on their own, while building on top of your work.

Finally, experiments are useful to show your work to any reviewer. This is useful to allow other people to review your work, and ensure you have not missed anything and that your models were built based on solid methods as well as verifying you have considered other options in your work. In some cases this would be a model governance organization whose job is to verify your work. In other instances, it may just be your peer who reviews your work as part of a collaboration effort.

Back to top

1.9.2 Artifact

Artifacts are stored in the project and are versioned. For more information, see data management and versioning

1.9.3 Tasks

A task can be created from a template, and can run over different runtimes or functions. Therefore, you can define a task once and reuse it in different scenarios. For instance, you can define a task with some parameters and inputs datasets, and call a local function, or use the same task to call a distributed job.

For example:

```
run = run_local(task,
                command='training.py')
```

Moving from local notebook execution to remote execution — such as running a container job, a scaled-out framework, or an automated workflow engine like Kubeflow Pipelines — is seamless: just swap the runtime function or wire functions in a graph. Continuous build integration and deployment (CI/CD) steps can also be configured as part of the workflow, using the `deploy_step` function method.

Tasks also support the `.with_hyper_param` method, which iterates over different parameter values in a single command For instance, the following command calls `training.py` with a task that iterates through different values for `p2`:

```
run = run_local(task.with_hyper_params({'p2': [5, 2, 3]}, 'min.loss'),
                command='training.py')
```

Back to top

1.9.4 Managed and Portable Execution

MLRun supports various types of **"runtimes"** — computation frameworks such as local, Kubernetes job, Dask, Nuclio, Spark, or MPI job (Horovod). Runtimes may support parallelism and clustering to distribute the work among multiple workers (processes/containers).

The following code example creates a task that defines a run specification — including the run parameters, inputs, and secrets. You run the task on a "job" function, and print the result output (in this case, the "model" artifact) or watch the run's progress. For more information and examples, see the **Examples section**.

```
# Create a task and set its attributes
task = NewTask(handler=handler, name='demo', params={'p1': 5})
task.with_secrets('file', 'secrets.txt').set_label('type', 'demo')

run = new_function(command='myfile.py', kind='job').run(task)
run.logs(watch=True)
run.show()
print(run.artifact('model'))
```

Back to top

1.9.5 Functions

A function is a software package with one or more methods and runtime-specific attributes (such as image, command, arguments, and environment). A function can run one or more runs or tasks

Functions can be created from an existing template, for instance, check out the MLRun Functions Marketplace. This marketplace is a centralized location for open-source contributions of function components that are commonly used in machine-learning development.

Functions (function objects) can be created by using any of the following methods:

- **`new_function`** — creates a function "from scratch" or from another function.
- **`code_to_function`** — creates a function from local or remote source code or from a web notebook.
- **`import_function`** — imports a function from a local or remote YAML function-configuration file or from a function object in the MLRun database (using a DB address of the format `db://<project>/<name>[:<tag>]`).

You can use the `save` function method to save a function object in the MLRun database, or the `export` method to save a YAML function-configuration function to your preferred local or remote location. For function-method details and examples, see the embedded documentation/help text.

Functions are stored in the project and are versioned. Therefore, you can always view previous code and go back to previous functions if needed.

Back to top

1.9.6 Jobs

Jobs contain all the information about functions that were executed within the project. You can view per each job the parameters and the results of that job.

You can view the artifacts produced by each job, for example, view the datasets that were produced by this particular job

Automated Parameterization, Artifact Tracking, and Logging

After running a job, you need to be able to track it, including viewing the run parameters, inputs, and outputs. To support this, MLRun introduces a concept of a runtime **"context"**: the code can be set up to get parameters and inputs from the context, as well as log run outputs, artifacts, tags, and time-series metrics in the context.

Example

The following code example from the **train-xgboost.ipynb** notebook of the MLRun XGBoost demo (**demo-xgboost**) defines two functions: the `iris_generator` function loads the Iris data set and saves it to the function's context object; the `xgb_train` function uses XGBoost to train an ML model on a data set and saves the log results in the function's context:

```
import xgboost as xgb
import os
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.metrics import accuracy_score
from mlrun.artifacts import PlotArtifact
import pandas as pd


def iris_generator(context):
    iris = load_iris()
    iris_dataset = pd.DataFrame(data=iris.data, columns=iris.feature_names)
    iris_labels = pd.DataFrame(data=iris.target, columns=['label'])
    iris_dataset = pd.concat([iris_dataset, iris_labels], axis=1)
    context.logger.info('Saving Iris data set to "{}"'.format(context.out_path))
    context.log_dataset('iris_dataset', df=iris_dataset)


def xgb_train(context,
              dataset='',
              model_name='model.bst',
              max_depth=6,
              num_class=10,
              eta=0.2,
              gamma=0.1,
              steps=20):

    df = pd.read_csv(dataset)
    X = df.drop(['label'], axis=1)
    y = df['label']

    X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.2)
    dtrain = xgb.DMatrix(X_train, label=Y_train)
```

(continues on next page)

(continued from previous page)

```
    dtest = xgb.DMatrix(X_test, label=Y_test)

    # Get parameters from event
    param = {"max_depth": max_depth,
             "eta": eta, "nthread": 4,
             "num_class": num_class,
             "gamma": gamma,
             "objective": "multi:softprob"}

    xgb_model = xgb.train(param, dtrain, steps)

    preds = xgb_model.predict(dtest)
    best_preds = np.asarray([np.argmax(line) for line in preds])

    context.log_result('accuracy', float(accuracy_score(Y_test, best_preds)))
    context.log_artifact('model', body=bytes(xgb_model.save_raw()),
                         local_path=model_name, labels={'framework': 'xgboost'})
```

The example training function can be executed locally with parameters, and the run results and artifacts can be logged automatically into a database by using a single command, as demonstrated in the following example; the example sets the function's `eta` parameter:

```
train_run = run_local(handler=xgb_train, pramas={'eta': 0.3})
```

Alternatively, you can replace the function with a serverless runtime to run the same code on a remote cluster, which could result in a ~10x performance boost. You can find examples for different runtimes — such as a Kubernetes job, Nuclio, Dask, Spark, or an MPI job — in the MLRun **examples** directory.

If you run your code from the `main` function, you can get the runtime context by calling the `get_or_create_ctx` method, as demonstrated in the following code from the MLRun **training.py** example application. The code also demonstrates how you can use the context object to read and write execution metadata, parameters, secrets, inputs, and outputs:

```
from mlrun import get_or_create_ctx
from mlrun.artifacts import ChartArtifact
import pandas as pd


def my_job(context, p1=1, p2='x'):
    # load MLRUN runtime context (will be set by the runtime framework e.g. KubeFlow)

    # get parameters from the runtime context (or use defaults)

    # access input metadata, values, files, and secrets (passwords)
    print(f'Run: {context.name} (uid={context.uid})')
    print(f'Params: p1={p1}, p2={p2}')
    print('accesskey = {}'.format(context.get_secret('ACCESS_KEY')))
    print('file\n{}\n'.format(context.get_input('infile.txt', 'infile.txt').get()))

    # Run some useful code e.g. ML training, data prep, etc.

    # log scalar result values (job result metrics)
    context.log_result('accuracy', p1 * 2)
    context.log_result('loss', p1 * 3)
    context.set_label('framework', 'sklearn')
```

(continues on next page)

(continued from previous page)

```
    # log various types of artifacts (file, web page, table), will be versioned and␣
↪visible in the UI
    context.log_artifact('model', body=b'abc is 123', local_path='model.txt', labels={
↪'framework': 'xgboost'})
    context.log_artifact('html_result', body=b'<b> Some HTML <b>', local_path='result.
↪html')

    # create a chart output (will show in the pipelines UI)
    chart = ChartArtifact('chart')
    chart.labels = {'type': 'roc'}
    chart.header = ['Epoch', 'Accuracy', 'Loss']
    for i in range(1, 8):
        chart.add_row([i, i/20+0.75, 0.30-i/20])
    context.log_artifact(chart)

    raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
                'last_name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze'],
                'age': [42, 52, 36, 24, 73],
                'testScore': [25, 94, 57, 62, 70]}
    df = pd.DataFrame(raw_data, columns=[
        'first_name', 'last_name', 'age', 'testScore'])
    context.log_dataset('mydf', df=df, stats=True)


if __name__ == "__main__":
    context = get_or_create_ctx('train')
    p1 = context.get_param('p1', 1)
    p2 = context.get_param('p2', 'a-string')
    my_job(context, p1, p2)
```

The example **training.py** application can be invoked as a local task, as demonstrated in the following code from the MLRun **Examples section**:

```
run = run_local(task, command='training.py')
```

Alternatively, you can invoke the application by using the `mlrun` CLI; edit the parameters, inputs, and/or secret information, as needed, and ensure that **training.py** is found in the execution path or edit the file path in the command:

```
mlrun run --name train -p p2=5 -i infile.txt=s3://my-bucket/infile.txt -s␣
↪file=secrets.txt training.py
```

Back to top

Using Hyperparameters for Job Scaling

Data science involves long computation times and data-intensive tasks. To ensure efficiency and scalability, you need to implement parallelism whenever possible. MLRun supports this by using two mechanisms:

1. Clustering — run the code on a distributed processing engine (such as Dask, Spark, or Horovod).
2. Load-balancing/partitioning — split (partition) the work across multiple workers.

MLRun functions and tasks can accept hyperparameters or parameter lists, deploy many parallel workers, and partition the work among the deployed workers. The parallelism implementation is left to the runtime. Each runtime may have its own method of concurrent tasks execution. For example, the Nuclio serverless engine manages many micro threads in the same process, which can run multiple tasks in parallel. In a containerized system like Kubernetes, you can launch multiple containers, each processing a different task.

MLRun supports parallelism. For example, the following code demonstrates how to use hyperparameters to run the XGBoost model-training task from the example in the previous section (`xgb_train`) with different parameter combinations:

```
    parameters = {
          "eta":       [0.05, 0.10, 0.20, 0.30],
          "max_depth": [3, 4, 5, 6, 8, 10],
          "gamma":     [0.0, 0.1, 0.2, 0.3],
          }

    task = NewTask(handler=xgb_train, out_path='/User/mlrun/data').with_hyper_
↪params(parameters, 'max.accuracy')
    run = run_local(task)
```

This code demonstrates how to instruct MLRun to run the same task while choosing the parameters from multiple lists (grid search). MLRun then records all the runs, but marks only the run with minimal loss as the selected result. For parallelism, it would be better to use runtimes like Dask, Nuclio, or jobs.

Alternatively, you can run a similar task (with hyperparameters) by using the MLRun CLI (`mlrun`); ensure that **training.py** is found in the execution path or edit the file path in the command:

```
mlrun run --name train_hyper -x p1="[3,7,5]" -x p2="[5,2,9]" --out-path '/User/mlrun/
↪data' training.py
```

You can also use a parameters file if you want to control the parameter combinations or if the parameters are more complex. The following code from the **Examples section** demonstrates how to run a task that uses a CSV parameters file (**params.csv** in the current directory):

```
    task = NewTask(handler=xgb_train).with_param_file('params.csv', 'max.accuracy')
    run = run_local(task)
```

> **Note:** Parameter lists can be used in various ways. For example, you can pass multiple parameter files and use multiple workers to process the files simultaneously instead of one at a time.

Back to top

Automated Code Deployment and Containerization

MLRun adopts Nuclio serverless technologies for automatically packaging code and building containers. This enables you to provide code with some package requirements and let MLRun build and deploy your software.

To build or deploy a function, all you need is to call the function's `deploy` method, which initiates a build or deployment job. Deployment jobs can be incorporated in pipelines just like regular jobs (using the `deploy_step` method of the function or Kubernetes-job runtime), thus enabling full automation and CI/CD.

A functions can be built from source code or from a function specification, web notebook, Git repo, or TAR archive.

A function can also be built by using the `mlrun` CLI and providing it with the path to a YAML function-configuration file. You can generate such a file by using the `to_yaml` or `export` function method. For example, the following CLI code builds a function from a **function.yaml** file in the current directory:

```
mlrun build function.yaml
```

Following is an example **function.yaml** configuration file:

```
kind: job
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: remote-git-test
  project: default
  tag: latest
spec:
  command: 'myfunc.py'
  args: []
  image_pull_policy: Always
  build:
    commands: ['pip install pandas']
    base_image: mlrun/mlrun:dev
    source: git://github.com/mlrun/ci-demo.git
```

For more examples of building and running functions remotely using the MLRun CLI, see the **remote** example.

You can also convert your web notebook to a containerized job, as demonstrated in the following sample code; for a similar example with more details, see the **mlrun_jobs.ipynb** example:

```
# Create an ML function from the notebook code and annotations, and attach a
# v3io Iguazio Data Science Platform data volume to the function
fn = code_to_function(kind='job').apply(mount_v3io())

# Prepare an image from the dependencies to allow updating the code and
# parameters per run without the need to build a new image
fn.build(image='mlrun/nuctest:latest')
```

Back to top

1.9.7 Pipelines

Pipelines are reusable end-to-end ML workflows. MLRun enables you to run your functions while saving outputs and artifacts in a way that is visible to **Kubeflow Pipelines**, which enable:

- End to end orchestration: enabling and simplifying the orchestration of end to end machine learning pipelines
- Easy experimentation: making it easy for you to try numerous ideas and techniques, and manage your various trials/experiments.
- Easy re-use: enabling you to re-use components and pipelines to quickly cobble together end to end solutions, without having to re-build each time.

For an example of a full ML pipeline that's implemented in a web notebook, see the Sklearn MLRun demo (**demo-sklearn-project**). The **sklearn-project.ipynb** demo notebook includes the following code for implementing an ML-training pipeline:

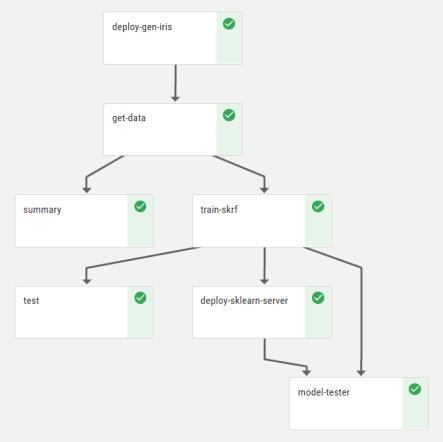
```
from kfp import dsl
from mlrun import mount_v3io

funcs = {}
DATASET = 'iris_dataset'
LABELS  = "label"

def init_functions(functions: dict, project=None, secrets=None):
    for f in functions.values():
        f.apply(mount_v3io())
        f.spec.image_pull_policy = 'Always'
```

(continues on next page)

(continued from previous page)

```
@dsl.pipeline(
    name="My XGBoost training pipeline",
    description="Shows how to use mlrun."
)
def kfpipeline():

    # build our ingestion function (container image)
    builder = funcs['gen-iris'].deploy_step(skip_deployed=True)

    # run the ingestion function with the new image and params
    ingest = funcs['gen-iris'].as_step(
        name="get-data",
        handler='iris_generator',
        image=builder.outputs['image'],
        params={'format': 'pq'},
        outputs=[DATASET])

    # analyze our dataset
    describe = funcs["describe"].as_step(
        name="summary",
        params={"label_column": LABELS},
        inputs={"table": ingest.outputs[DATASET]})

    # train with hyper-paremeters
    train = funcs["train"].as_step(
        name="train-skrf",
        params={"model_pkg_class" : "sklearn.ensemble.RandomForestClassifier",
                "sample"          : -1,
                "label_column"    : LABELS,
                "test_size"       : 0.10},
        hyperparams={'CLASS_n_estimators': [100, 300, 500]},
        selector='max.accuracy',
        inputs={"dataset"         : ingest.outputs[DATASET]},
        outputs=['model', 'test_set'])

    # test and visualize our model
    test = funcs["test"].as_step(
        name="test",
        params={"label_column": LABELS},
        inputs={"models_path" : train.outputs['model'],
                "test_set"    : train.outputs['test_set']})

    # deploy our model as a serverless function
    deploy = funcs["serving"].deploy_step(models={f"{DATASET}_v1": train.outputs[
↪'model']})
```

Visually, the workflow would look as follows:



Back to top

1.9.8 Viewing Run Data and Performing Database Operations

When you configure an MLRun database, the results, parameters, and input and output artifacts of each run are recorded in the database. You can view the results and perform operations on the database by using either of the following methods:

- Using *the MLRun dashboard*
- Using *DB methods* from your code

Back to top

The MLRun Dashboard

The MLRun dashboard is a graphical user interface (GUI) for working with MLRun and viewing run data.

Back to top

MLRun Database Methods

You can use the `get_run_db` DB method to get an MLRun DB object for a configured MLRun database or API service. Then, use the DB object's `connect` method to connect to the database or API service, and use additional methods to perform different operations, such as listing run artifacts or deleting completed runs. For more information and examples, see the **mlrun_db.ipynb** example notebook, which includes the following sample DB method calls:

```
from mlrun import get_run_db

# Get an MLRun DB object and connect to an MLRun database/API service.
# Specify the DB path (for example, './' for the current directory) or
# the API URL ('http://mlrun-api:8080' for the default configuration).
db = get_run_db('./').connect()

# List all runs
db.list_runs('').show()

# List all artifacts for version 'latest' (default)
db.list_artifacts('', tag='').show()
```

(continues on next page)

(continued from previous page)

```
# Check different artifact versions
db.list_artifacts('ch', tag='*').show()

# Delete completed runs
db.del_runs(state='completed')
```

1.9.9 Additional Information and Examples

Replacing Runtime Context Parameters from the CLI

You can use the MLRun CLI (`mlrun`) to run MLRun functions or code and change the parameter values.

For example, the following CLI command runs the example XGBoost training code from the previous tutorial examples:

```
python -m mlrun run -p p1=5 -s file=secrets.txt -i infile.txt=s3://mybucket/infile.
↪txt training.py
```

When running this sample command, the CLI executes the code in the **training.py** application using the provided run information:

- The value of parameter `p1` is set to `5`, overwriting the current parameter value in the run context.
- The file **infile.txt** is downloaded from a remote "mybucket" AWS S3 bucket.
- The credentials for the S3 download are retrieved from a **secrets.txt** file in the current directory.

Remote Execution

You can also run the same MLRun code that you ran locally as a remote HTTP endpoint.

Nuclio Example

For example, you can wrap the XGBoost training code from the previous tutorial examples within a serverless Nuclio handler function, and execute the code remotely using a similar CLI command to the one that you used locally.

You can run the following code from a Jupyter Notebook to create a Nuclio function from the notebook code and annotations, and deploy the function to a remote cluster.

Note:

- Before running the code, install the `nuclio-jupyter` package for using Nuclio from Jupyter Notebook.
- The example uses `apply(mount_v3io()` to attach a v3io Iguazio Data Science Platform datastore volume to the function. By default, the v3io mount mounts the home directory of the platform's running user into the `\\User` function path.

```
# Create an `xgb_train` Nuclio function from the notebook code and annotations;
# add a v3io data volume and a multi-worker HTTP trigger for parallel execution
fn = code_to_function('xgb_train', runtime='nuclio:mlrun')
fn.apply(mount_v3io()).with_http(workers=32)

# Deploy the function
run = fn.run(task, handler='xgb_train')
```

To execute the code remotely, run the same CLI command as in the previous tutorial examples and just substitute the code file name at the end with your function's URL. For example, run the following command and replace `<function endpoint>` with your remote function endpoint:

```
mlrun run -p p1=5 -s file=secrets.txt -i infile.txt=s3://mybucket/infile.txt http://
↪<function-endpoint>
```

Back to top

1.9.10 Running an MLRun Service

An MLRun service is a web service that manages an MLRun database for tracking and logging MLRun run information, and exposes an HTTP API for working with the database and performing MLRun operations.

You can create and run an MLRun service by using either of the following methods:

- *Using Docker*
- *Using the MLRun CLI*

  Note: For both methods, you can optionally configure the service port and/or directory path by setting the `MLRUN_httpdb__port` and `MLRUN_httpdb__dirpath` environment variables instead of the respective run parameters or CLI options.

Using the MLRun CLI to Run an MLRun Service

Use the `db` command of the MLRun CLI (`mlrun`) to create and run an instance of the MLRun service from the command line:

```
mlrun db [OPTIONS]
```

To see the supported options, run `mlrun db --help`:

```
Options:
  -p, --port INTEGER  HTTP port for serving the API
  -d, --dirpath TEXT  Path to the MLRun service directory
```

1.10 Model Management and Serving

1.10.1 Deploy

REST API Endpoint

Model deployment can be done by the combination of MLRun and Nuclio. Nuclio is a serverless frameworks that enable users to run functions in a scalable way on a distributed cluster. MLRun is integrated in a way that it provides an easy way for developers to take python code along with its model and convert it into a Nuclio function with minimal DevOps. By default, the Nuclio function is being created with an HTTP trigger so it has an endpoint that receive HTTP requests. In the functions marketplace you'll find several serving functions that can be used out of the box. Below you can find an example of deploying a serving function from the marketplace. Make sure to set the models_path with your model.

```
fn = project.import_function('hub://model_server', 'serving')
fn.set_envs({'SERVING_MODEL_iris_dataset_v1': models_path})
fn.deploy()
```

The built-in serving functions can capture the features that are being sent to the model along with the actual results and store them as a stream for monitoring and drift detection. In order to do that one can use the `INFERENCE_STREAM` parameters and provide the path to the streaming location.

```
fn.set_envs({'SERVING_MODEL_iris_dataset_v1': models_path,
             'INFERENCE_STREAM': 'users/admin/tststream'})
fn.deploy()
```

In some cases, there is a need to select a specific function version for alignment with the framework the user is working on. Here is an example of working with different TensorFlow frameworks:

```
if tf_ver == 'v1':
    project.set_function('hub://tf1_serving', 'serving')
else:
    project.set_function('hub://tf2_serving', 'serving')
```

Streaming

TBD

Batch Prediction

TBD

1.10.2 Test

Note, that once the model has been deployed, the best practice is to test the endpoint and validate that the model is working properly. In order to do that, you can also leverage the marketplace for getting those test functions. It basically takes several records from the test data and sends it over to the deployed function to validate that the model is working. When working with the Iguazio Data Science Platform, those serving functions (aka Nuclio functions) can be viewed in the Iguazio dashboard under the projects tab. when working with Nuclio standalone you can view it in the Nuclio dashboard.

```
project.import_function('hub://model_server_tester', 'live_tester')
```

1.10.3 Monitoring and Drift Detection

After the model has been deployed, users need a way to monitor the running model to make sure the model is working properly. As we've seen in the deploy section, the best practice is to write the model's features and results to a stream. From there, we can run a drift analysis, show it in a real time dashboard and store data for historical analysis. In the functions marketplace you can find a drift detection function (AKA concept-drift) that reads the stream data, analyze if there is a drift and write it down to a time series database for real time dashboarding and also to a parquet file for historical analysis. Here is an example of such pipeline with the concept drift function:

1.11 Examples

MLRun has many code examples and tutorial Jupyter notebooks with embedded documentation, ranging from examples of basic tasks to full end-to-end use-case applications, including the following; note that some of the examples are found in other mlrun GitHub repositories:

- Learn MLRun basics — **mlrun_basics.ipynb**
- Convert local runs to Kubernetes jobs and create automated pipelines in a single notebook — **mlrun_jobs.ipynb**
- End-to-end ML pipeline— **demo-sklearn-project**, including:
  - Data ingestion and analysis
  - Model training
  - Verification
  - Model deployment
- MLRun with scale-out runtimes —
  - Distributed TensorFlow with Horovod and MPIJob — **horovod-project.ipynb**
  - Serverless model serving with Nuclio — **xgb_serving.ipynb**
  - Dask — **mlrun_dask.ipynb**

  - Spark — **mlrun_sparkk8s.ipynb**
- MLRun project and Git lifecycle —
  - Load a project from a remote Git location and run pipelines — **load-project.ipynb**
  - Create a new project, functions, and pipelines, and upload to Git — **new-project.ipynb**
- Import and export functions using files or Git — **mlrun_export_import.ipynb**
- Query the MLRun DB — **mlrun_db.ipynb**

1.11.1 Additional Examples

- Deep-learning pipeline (full end-to-end application), including data collection and labeling, model training and serving, and implementation of an automated workflow — mlrun/demo-image-classification repo
- Additional end-to-end use-case applications — mlrun/demos repo
- MLRun functions Library — mlrun/functions repo

Back to top

1.12 API

MLRun is organized into the following modules. The most common functions are exposed in the `mlrun` module, so we recommend starting there.

1.12.1 mlrun

MLRun is a generic and convenient mechanism for data scientists and software developers to describe and run tasks related to machine learning (ML) in various, scalable runtime environments and ML pipelines while automatically tracking executed code, metadata, inputs, and outputs. MLRun integrates with the Nuclio serverless project and with Kubeflow Pipelines.

The MLRun package (`mlrun`) includes a Python API library and the `mlrun` command-line interface (CLI).

class mlrun.**DataItem**(*key: str*, *store: mlrun.datastore.base.DataStore*, *subpath: str*, *url: str = ''*, *meta=None*, *artifact_url=None*)

Data input/output class abstracting access to various local/remote data sources

Attributes

artifact_url DataItem artifact url (when its an artifact) or url for simple dataitems

key DataItem key

kind DataItem store kind (file, s3, v3io, ..)

meta Artifact Metadata, when the DataItem is read from the artifacts store

store DataItem store object

suffix DataItem suffix (file extension) e.g.

url DataItem url e.g.

Methods

as_df(self[, columns, df_module, format])	return a dataframe object (generated from the dataitem).
download(self, target_path)	download to the target dir/path
get(self[, size, offset])	read all or a range and return thge content
listdir(self)	return a list of child file names
local(self)	get the local path of the file, download to tmp first if its a remote object
put(self, data[, append])	write/upload the data, append is only supported by some datastores
stat(self)	return FileStats class (size, modified, content_type)
upload(self, src_path)	upload the source file (src_path)

property artifact_url
DataItem artifact url (when its an artifact) or url for simple dataitems

as_df(*self*, *columns=None*, *df_module=None*, *format=''*, ***kwargs*)
return a dataframe object (generated from the dataitem).

Parameters

- **columns** – optional, list of columns to select
- **df_module** – optional, dataframe class (e.g. pd, dd, cudf, ..)
- **format** – file format, if not specified it will be deducted from the suffix

download(*self*, *target_path*)
download to the target dir/path

get(*self*, *size=None*, *offset=0*)
read all or a range and return thge content

property key
DataItem key

property kind
DataItem store kind (file, s3, v3io, ..)

listdir(*self*)
return a list of child file names

local(*self*)
get the local path of the file, download to tmp first if its a remote object

property meta
Artifact Metadata, when the DataItem is read from the artifacts store

put(*self*, *data*, *append=False*)
write/upload the data, append is only supported by some datastores

stat(*self*)
return FileStats class (size, modified, content_type)

property store
DataItem store object

property suffix
DataItem suffix (file extension) e.g. '.png'

upload(*self*, *src_path*)
upload the source file (src_path)

property url
DataItem url e.g. /dir/path, s3://bucket/path

class mlrun.**MLClientCtx**(*autocommit=False*, *tmp=''*, *log_stream=None*)
ML Execution Client Context

the context is generated using the get_or_create_ctx call (see its doc) and provides an interface to use run params, metadata, inputs, and outputs

base metadata include: uid, name, project, and iteration (for hyper params) users can set labels and annotations using set_labels(), set_annotation() access parameters and secrets using get_param(), get_secret() access input data objects using get_input() store results, artifacts, and real-time metrics using log_xx methods

see doc for the individual params and methods

Attributes

annotations dictionary with annotations (read-only)

artifacts dictionary of artifacts (read-only)

in_path default input path for data objects

inputs dictionary of input data items (read-only)

iteration child iteration index, for hyper parameters

labels dictionary with labels (read-only)

log_level get the logging level, e.g. 'debug', 'info', 'error'

logger built-in logger interface

out_path default output path for artifacts

parameters dictionary of run parameters (read-only)

project project name, runs can be categorized by projects

results dictionary of results (read-only)

tag run tag (uid or workflow id if exists)

uid Unique run id

Methods

artifact_subpath(self, *subpaths)	subpaths under output path artifacts path
commit(self, message)	save run state and add a commit message
commit_children(self[, best_run])	update/commit all children, and optionally mark the best note: best_run marks the child iteration number (starts from 1)
from_dict(attrs[, rundb, autocommit, tmp, ...])	create execution context from dict
get_child(self, **params)	get child context (iteration)
get_dataitem(self, url)	get mlrun dataitem from url
get_input(self, key, url)	get an input data object, data objects have methods such as .get(), .download(), .url, ..
get_meta(self)	Reserved for internal use

Continued on next page

Table 2 – continued from previous page

get_param(self, key[, default])	get a run parameter, or use the provided default if not set
get_secret(self, key)	get a key based secret e.g.
log_artifact(self, item[, body, local_path, ...])	log an output artifact and optionally upload it to data-store
log_dataset(self, key, df[, tag, ...])	log a dataset artifact and optionally upload it to data-store
log_iteration_results(self, best, summary, task)	Reserved for internal use
log_metric(self, key, value[, timestamp, labels])	TBD, log a real-time time-series metric
log_metrics(self, keyvals[, timestamp, labels])	TBD, log a set of real-time time-series metrics
log_model(self, key[, body, framework, tag, ...])	log a model artifact and optionally upload it to data-store
log_result(self, key, value[, commit])	log a scalar result value
log_results(self, results[, commit])	log a set of scalar result values
set_annotation(self, key, value, replace)	set/record a specific annotation
set_hostname(self, host)	update the hostname
set_label(self, key, value, replace)	set/record a specific label
set_state(self, state, error[, commit])	modify and store the run state or mark an error
to_dict(self)	convert the run context to a dictionary
to_json(self)	convert the run context to a json buffer
to_yaml(self)	convert the run context to a yaml buffer

set_logger_stream

property annotations
dictionary with annotations (read-only)

artifact_subpath(*self*, **subpaths*)
subpaths under output path artifacts path

property artifacts
dictionary of artifacts (read-only)

commit(*self*, *message: str = ''*)
save run state and add a commit message

commit_children(*self*, *best_run=0*)
update/commit all children, and optionally mark the best note: best_run marks the child iteration number (starts from 1)

classmethod from_dict(*attrs: dict*, *rundb=''*, *autocommit=False*, *tmp=''*, *host=None*, *log_stream=None*)
create execution context from dict

get_child(*self*, ***params*)
get child context (iteration)

allow sub experiments (epochs, hyper-param, ..) under a parent will create a new iteration, log_xx will update the child only use commit_children() to save all the children and specify the best run

Parameters ****params** – extra (or override) params to parent context

get_dataitem(*self*, *url*)
get mlrun dataitem from url

get_input(*self*, *key: str*, *url: str = ''*)
get an input data object, data objects have methods such as .get(), .download(), .url, .. to access the actual data

get_meta(*self*)
Reserved for internal use

get_param(*self*, *key: str*, *default=None*)
get a run parameter, or use the provided default if not set

get_secret(*self*, *key: str*)
get a key based secret e.g. DB password from the context secrets can be specified when invoking a run through files, env, ..

property in_path
default input path for data objects

property inputs
dictionary of input data items (read-only)

property iteration
child iteration index, for hyper parameters

property labels
dictionary with labels (read-only)

log_artifact(*self*, *item*, *body=None*, *local_path=None*, *artifact_path=None*, *tag=''*, *viewer=None*, *target_path=''*, *src_path=None*, *upload=None*, *labels=None*, *format=None*, *db_key=None*, ***kwargs*)
log an output artifact and optionally upload it to datastore

Parameters

- **item** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use:

  artifact_path=context.artifact_subpath('data')
- **tag** – version tag
- **viewer** – kubeflow viewer type
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **src_path** – deprecated, use local_path
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_dataset(*self*, *key*, *df*, *tag=''*, *local_path=None*, *artifact_path=None*, *upload=True*, *labels=None*, *format=''*, *preview=None*, *stats=False*, *db_key=None*, *target_path=''*, *extra_data=None*, ***kwargs*)
log a dataset artifact and optionally upload it to datastore

Parameters

- **key** – artifact key
- **df** – dataframe object
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use:

  artifact_path=context.artifact_subpath('data')
- **tag** – version tag
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **preview** – number of lines to store as preview in the artifact metadata
- **stats** – calculate and store dataset stats in the artifact metadata
- **extra_data** – key/value list of extra files/charts to link with this dataset
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_iteration_results(*self*, *best*, *summary: list*, *task: dict*, *commit=False*)
Reserved for internal use

property log_level
get the logging level, e.g. 'debug', 'info', 'error'

log_metric(*self*, *key: str*, *value*, *timestamp=None*, *labels=None*)
TBD, log a real-time time-series metric

log_metrics(*self*, *keyvals: dict*, *timestamp=None*, *labels=None*)
TBD, log a set of real-time time-series metrics

log_model(*self*, *key*, *body=None*, *framework=''*, *tag=''*, *model_dir=None*, *model_file=None*, *metrics=None*, *parameters=None*, *artifact_path=None*, *upload=True*, *labels=None*, *inputs=None*, *outputs=None*, *extra_data=None*, *db_key=None*)
log a model artifact and optionally upload it to datastore

Parameters

- **key** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **model_file** – path to the local model file we upload (seel also model_dir)
- **model_dir** – path to the local dir holding the model file and extra files
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use:

  artifact_path=context.artifact_subpath('data')
- **framework** – name of the ML framework

- **tag** – version tag
- **metrics** – key/value dict of model metrics
- **parameters** – key/value dict of model parameters
- **inputs** – ordered list of model input features (name, type, ..)
- **outputs** – ordered list of model output/result elements (name, type, ..)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **extra_data** – key/value list of extra files/charts to link with this dataset value can be abs/relative path string | bytes | artifact object
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_result (*self*, *key: str*, *value*, *commit=False*)
log a scalar result value

Parameters

- **key** – result key
- **value** – result value
- **commit** – commit (write to DB now vs wait for the end of the run)

log_results (*self*, *results: dict*, *commit=False*)
log a set of scalar result values

Parameters

- **results** – key/value dict or results
- **commit** – commit (write to DB now vs wait for the end of the run)

property logger
built-in logger interface

property out_path
default output path for artifacts

property parameters
dictionary of run parameters (read-only)

property project
project name, runs can be categorized by projects

property results
dictionary of results (read-only)

set_annotation (*self*, *key: str*, *value*, *replace: bool = True*)
set/record a specific annotation

set_hostname (*self*, *host: str*)
update the hostname

set_label (*self*, *key: str*, *value*, *replace: bool = True*)
set/record a specific label

set_state(*self*, *state: str = None*, *error: str = None*, *commit=True*)
modify and store the run state or mark an error

Parameters

- **state** – set run state
- **error** – error message (if exist will set the state to error)
- **commit** – will immediately update the state in the DB

property tag
run tag (uid or workflow id if exists)

to_dict(*self*)
convert the run context to a dictionary

to_json(*self*)
convert the run context to a json buffer

to_yaml(*self*)
convert the run context to a yaml buffer

property uid
Unique run id

mlrun.**NewTask**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *tuning_strategy=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*)
Creates a new task - see new_task

class mlrun.**RunObject**(*spec: mlrun.model.RunSpec = None*, *metadata: mlrun.model.RunMetadata = None*, *status: mlrun.model.RunStatus = None*)
A run

Attributes

metadata

outputs

spec

status

Methods

with_secrets(self, kind, source)	register a secrets source (file, env or dict)

artifact	
copy	
from_dict	
from_template	
logs	
output	
set_label	
show	
state	
to_dict	
to_env	
to_json	
to_str	
to_yaml	
uid	
with_hyper_params	
with_input	
with_param_file	
with_params	

class mlrun.**RunTemplate**(*spec: mlrun.model.RunSpec = None*, *metadata: mlrun.model.RunMetadata = None*)

Run template

Attributes

metadata

spec

Methods

with_secrets(self, kind, source)	register a secrets source (file, env or dict)

copy	
from_dict	
set_label	
to_dict	
to_env	
to_json	
to_str	
to_yaml	
with_hyper_params	
with_input	
with_param_file	
with_params	

with_secrets(*self*, *kind*, *source*)

register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, e.g.

proj.with_secrets('file', 'file.txt') proj.with_secrets('inline', {'key': 'val'}) proj.with_secrets('env', 'ENV1,ENV2')

Parameters

- **kind** – secret type (file, inline, env)
- **source** – secret data or link (see example)

Returns project object

mlrun.**code_to_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *filename: str = ''*, *handler: str = ''*, *kind: str = ''*, *image: str = None*, *code_output=''*, *embed_code=True*, *description=''*, *categories: list = None*, *labels: dict = None*, *with_doc=True*)

convert code or notebook to function object with embedded code code stored in the function spec and can be refreshed using .with_code() eliminate the need to build container images every time we edit the code

Parameters

- **name** – function name
- **project** – function project (none for 'default')
- **tag** – function tag (none for 'latest')
- **filename** – blank for current notebook, or path to .py/.ipynb file
- **handler** – name of function handler (if not main)
- **kind** – optional, runtime type local, job, dask, mpijob, ..
- **image** – optional, container image
- **code_output** – save the generated code (from notebook) in that path
- **embed_code** – embed the source code into the function spec
- **description** – function description
- **categories** – list of categories (for function marketplace)
- **labels** – dict of label names and values to tag the function
- **with_doc** – document the function parameters

Returns function object

mlrun.**function_to_module**(*code=''*, *workdir=None*, *secrets=None*)

Load code, notebook or mlrun function as .py module this function can import a local/remote py file or notebook or load an mlrun function object as a module, you can use this from your code, notebook, or another function (for common libs)

Note: the function may have package requirements which must be satisfied

example:

mod = mlrun.function_to_module('./examples/training.py') task = mlrun.new_task(inputs={'infile.txt': '../examples/infile.txt'}) context = mlrun.get_or_create_ctx('myfunc', spec=task) mod.my_job(context, p1=1, p2='x') print(context.to_yaml())

fn = mlrun.import_function('hub://open_archive') mod = mlrun.function_to_module(fn) data = mlrun.run.get_dataitem("https://fpsignals-public.s3.amazonaws.com/catsndogs.tar.gz") context = mlrun.get_or_create_ctx('myfunc') mod.open_archive(context, archive_url=data) print(context.to_yaml())

Parameters

- **code** – path/url to function (.py or .ipynb or .yaml) OR function object
- **workdir** – code workdir
- **secrets** – secrets needed to access the URL (e.g.s3, v3io, ..)

Returns python module

mlrun.**get_object**(*url*, *secrets=None*, *size=None*, *offset=0*, *db=None*)
get mlrun dataitem body (from path/url)

mlrun.**get_or_create_ctx**(*name: str*, *event=None*, *spec=None*, *with_env: bool = True*, *rundb: str = ''*)
called from within the user program to obtain a run context

the run context is an interface for receiving parameters, data and logging run results, the run context is read from the event, spec, or environment (in that order), user can also work without a context (local defaults mode)

all results are automatically stored in the "rundb" or artifact store, the path to the rundb can be specified in the call or obtained from env.

Parameters

- **name** – run name (will be overridden by context)
- **event** – function (nuclio Event object)
- **spec** – dictionary holding run spec
- **with_env** – look for context in environment vars, default True
- **rundb** – path/url to the metadata and artifact database

Returns execution context

Example:

load MLRUN runtime context (will be set by the runtime framework e.g. KubeFlow) context = get_or_create_ctx('train')

get parameters from the runtime context (or use defaults) p1 = context.get_param('p1', 1) p2 = context.get_param('p2', 'a-string')

access input metadata, values, files, and secrets (passwords) print(f'Run: {context.name} (uid={context.uid})') print(f'Params: p1={p1}, p2={p2}') print('accesskey = {}'.format(context.get_secret('ACCESS_KEY'))) print('file: {}'.format(context.get_input('infile.txt').get()))

RUN some useful code e.g. ML training, data prep, etc.

log scalar result values (job result metrics) context.log_result('accuracy', p1 * 2) context.log_result('loss', p1 * 3) context.set_label('framework', 'sklearn')

log various types of artifacts (file, web page, table), will be versioned and visible in the UI context.log_artifact('model.txt', body=b'abc is 123', labels={'framework': 'xgboost'}) context.log_artifact('results.html', body=b'<b> Some HTML <b>', viewer='web-app')

mlrun.**get_pipeline**(*run_id*, *namespace=None*)
Get Pipeline status

Parameters

- **run_id** – id of pipelines run
- **namespace** – k8s namespace if not default

Returns kfp run dict

mlrun.**get_run_db**(*url=''*)
Returns the runtime database

mlrun.**import_function**(*url=''*, *secrets=None*, *db=''*)
Create function object from DB or local/remote YAML file

Reading from a file or remote URL (http(s), s3, git, v3io, ..)

Parameters

- **url** – path/url to function YAML file or 'db://{project}/{name}[:tag]' when reading from mlrun db
- **secrets** – optional, credentials dict for DB or URL (s3, v3io, ...)
- **db** – optional, mlrun api/db path

Returns function object

mlrun.**load_project**(*context*, *url=None*, *name=None*, *secrets=None*, *init_git=False*, *subpath=''*, *clone=False*)
Load an MLRun project from git or tar or dir

Parameters

- **context** – project local directory path
- **url** – git or tar.gz sources archive path e.g.: git://github.com/mlrun/demo-xgb-project.git
- **name** – project name
- **secrets** – key:secret dict or SecretsStore used to download sources
- **init_git** – if True, will git init the context dir
- **subpath** – project subpath (within the archive)
- **clone** – if True, always clone (delete any existing content)

Returns project object

mlrun.**mount_v3io**(*name='v3io'*, *remote='~/'*, *mount_path='/User'*, *access_key=''*, *user=''*, *secret=None*)
Modifier function to apply to a Container Op to volume mount a v3io path

Parameters

- **name** – the volume name
- **remote** – the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/
- **mount_path** – the volume mount path
- **access_key** – the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used
- **user** – the username used to auth against v3io. if not given V3IO_USERNAME env var will be used
- **secret** – k8s secret name which would be used to get the username and access key to auth against v3io.

mlrun.**new_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *kind: str = ''*, *command: str = ''*, *image: str = ''*, *args: list = None*, *runtime=None*, *mode=None*, *kfp=None*)
Create a new ML function from base properties

e.g.: # define a container based function f = new_function(command='job://training.py -v', image='myrepo/image:latest')

define a handler function (execute a local function handler) f = new_function().run(task, handler=myfunction)

Parameters

- **name** – function name
- **project** – function project (none for 'default')
- **tag** – function version tag (none for 'latest')
- **kind** – runtime type (local, job, nuclio, spark, mpijob, dask, ..)
- **command** – command/url + args (e.g.: training.py –verbose)
- **image** – container image (start with '.' for default registry)
- **args** – command line arguments (override the ones in command)
- **runtime** – runtime (job, nuclio, spark, dask ..) object/dict store runtime specific details and preferences
- **mode** – runtime mode, e.g. noctx, pass to bypass mlrun
- **kfp** – reserved, flag indicating running within kubeflow pipeline

Returns function object

mlrun.**new_project**(*name*, *context=None*, *init_git=False*)

Create a new MLRun project

Parameters

- **name** – project name
- **context** – project local directory path
- **init_git** – if True, will git init the context dir

Returns project object

mlrun.**new_task**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *tuning_strategy=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*)

Creates a new task

Parameters

- **name** – task name
- **project** – task project

:param handler code entry-point/hanfler name :param params: input parameters (dict) :param hyper_params: dictionary of hyper parameters and list values, each

hyper param holds a list of values, the run will be executed for every parameter combination (GridSearch)

Parameters

- **param_file** – a csv file with parameter combinations, first row hold the parameter names, following rows hold param values
- **selector** – selection criteria for hyper params e.g. "max.accuracy"

- **tuning_strategy** – selection strategy for hyper params e.g. list, grid, random
- **inputs** – dictionary of input objects + optional paths (if path is omitted the path will be the in_path/key.
- **outputs** – dictionary of input objects + optional paths (if path is omitted the path will be the out_path/key.
- **in_path** – default input path/url (prefix) for inputs
- **out_path** – default output path/url (prefix) for artifacts
- **artifact_path** – default artifact output path
- **secrets** – extra secrets specs, will be injected into the runtime e.g. ['file=<filename>', 'env=ENV_KEY1,ENV_KEY2']
- **base** – task instance to use as a base instead of a fresh new task instance

mlrun.**run_local**(*task=None*, *command=''*, *name: str = ''*, *args: list = None*, *workdir=None*, *project: str = ''*, *tag: str = ''*, *secrets=None*, *handler=None*, *params: dict = None*, *inputs: dict = None*, *artifact_path: str = ''*)

Run a task on function/code (.py, .ipynb or .yaml) locally,

e.g.: # define a task task = new_task(params={'p1': 8}, out_path=out_path) # run run = run_local(spec, command='src/training.py', workdir='src')

or specify base task parameters (handler, params, ..) in the call

run = run_local(handler=my_function, params={'x': 5})

Parameters

- **task** – task template object or dict (see RunTemplate)
- **command** – command/url/function
- **name** – ad hook function name
- **args** – command line arguments (override the ones in command)
- **workdir** – working dir to exec in
- **project** – function project (none for 'default')
- **tag** – function version tag (none for 'latest')
- **secrets** – secrets dict if the function source is remote (s3, v3io, ..)
- **handler** – pointer or name of a function handler
- **params** – input parameters (dict)
- **inputs** – input objects (dict of key: path)
- **artifact_path** – default artifact output path

Returns run object

mlrun.**run_pipeline**(*pipeline*, *arguments=None*, *project=None*, *experiment=None*, *run=None*, *namespace=None*, *artifact_path=None*, *ops=None*, *url=None*, *ttl=None*)

remote KubeFlow pipeline execution

Submit a workflow task to KFP via mlrun API service

:param pipeline KFP pipeline function or path to .yaml/.zip pipeline file :param arguments pipeline arguments :param experiment experiment name :param run optional, run name :param namespace Kubernetes namespace

(if not using default) :param url optional, url to mlrun API service :param artifact_path target location/url for mlrun artifacts :param ops additional operators (.apply() to all pipeline functions) :param ttl pipeline ttl in secs (after that the pods will be removed)

Returns kubeflow pipeline id

mlrun.**set_environment**(*api_path: str = None*, *artifact_path: str = ''*, *project: str = ''*)
set and test default config for: api path, artifact_path and project

this function will try and read the configuration from the environment/api and merge it with the user provided parameters

Parameters

- **api_path** – location/url of mlrun api service
- **artifact_path** – path/url for storing experiment artifacts
- **project** – default project name

Returns

actual artifact path/url, can be used to create subpaths per task, e.g.: artifact_path = set_environment()

data_subpath = os.join(artifact_path, 'data')

mlrun.**v3io_cred**(*api=''*, *user=''*, *access_key=''*)
Modifier function to copy local v3io env vars to task Usage:

train = train_op(. . .) train.apply(use_v3io_cred())

mlrun.**wait_for_pipeline_completion**(*run_id*, *timeout=3600*, *expected_statuses: List[str] = None*, *namespace=None*)
Wait for Pipeline status, timeout in sec

Parameters

- **run_id** – id of pipelines run
- **timeout** – wait timeout in sec
- **expected_statuses** – list of expected statuses, one of [Succeeded | Failed | Skipped | Error], by default [Succeeded]
- **namespace** – k8s namespace if not default

Returns kfp run dict

1.12.2 mlrun.artifacts

mlrun.artifacts.**get_model**(*model_dir*, *suffix=''*)
return model file, model spec object, and list of extra data items

this function will get the model file, metadata, and extra data the returned model file is always local, when using remote urls

(such as v3io://, s3://, store://, ..) it will be copied locally.

returned extra data dict (of key, DataItem objects) allow reading additional model files/objects e.g. use DataItem.get() or .download(target) .as_df() to read

example: model_file, model_artifact, extra_data = get_model(models_path, suffix='.pkl') model = load(open(model_file, "rb")) categories = extra_data['categories'].as_df()

Parameters

- **model_dir** – model dir or artifact path (store://..) or DataItem
- **suffix** – model filename suffix (when using a dir)

Returns model filename, model artifact object, extra data dict

mlrun.artifacts.**update_model**(*model_artifact*, *parameters: dict = None*, *metrics: dict = None*, *extra_data: dict = None*, *inputs: list = None*, *outputs: list = None*, *key_prefix: str = ''*, *labels: dict = None*)

Update model object attributes

this method will edit or add attributes to a model object

example:

update_model(model_path, metrics={'speed': 100}, extra_data={'my_data': b'some text', 'file': 's3://mybucket/..'})

Parameters

- **model_artifact** – model artifact object or path (store://..) or DataItem
- **parameters** – parameters dict
- **metrics** – model metrics e.g. accuracy
- **extra_data** – extra data items key, value dict (value can be: path string | bytes | artifact)
- **inputs** – list of inputs (feature vector schema)
- **outputs** – list of outputs (output vector schema)
- **key_prefix** – key prefix to add to metrics and extra data items
- **labels** – metadata labels

1.12.3 mlrun.config

Configuration system.

Configuration can be in either a configuration file specified by MLRUN_CONFIG_FILE environment variable or by environmenet variables.

Environment variables are in the format "MLRUN_httpdb__port=8080". This will be mapped to config.httpdb.port. Values should be in JSON format.

class mlrun.config.**Config**(*cfg=None*)

Bases: object

Attributes

dbpath

kfp_image When this configuration is not set we want to set it to mlrun/mlrun, but we need to use the enrich_image method.

version

Methods

dump_yaml	
from_dict	
reload	
update	

property dbpath

dump_yaml(*self*, *stream=None*)

classmethod from_dict(*dict_*)

property kfp_image
When this configuration is not set we want to set it to mlrun/mlrun, but we need to use the enrich_image method. The problem is that the mlrun.utils.helpers module is importing the config (this) module, so we must import the module inside this function (and not on initialization), and then calculate this property value here.

static reload()

update(*self*, *cfg*)

property version

mlrun.config.**read_env**(*env=None*, *prefix='MLRUN_'*)
Read configuration from environment

1.12.4 mlrun.execution

class mlrun.execution.**MLClientCtx**(*autocommit=False*, *tmp=''*, *log_stream=None*)
Bases: object

ML Execution Client Context

the context is generated using the get_or_create_ctx call (see its doc) and provides an interface to use run params, metadata, inputs, and outputs

base metadata include: uid, name, project, and iteration (for hyper params) users can set labels and annotations using set_labels(), set_annotation() access parameters and secrets using get_param(), get_secret() access input data objects using get_input() store results, artifacts, and real-time metrics using log_xx methods

see doc for the individual params and methods

Attributes

annotations dictionary with annotations (read-only)

artifacts dictionary of artifacts (read-only)

in_path default input path for data objects

inputs dictionary of input data items (read-only)

iteration child iteration index, for hyper parameters

labels dictionary with labels (read-only)

log_level get the logging level, e.g. 'debug', 'info', 'error'

logger built-in logger interface

out_path default output path for artifacts

parameters dictionary of run parameters (read-only)

project project name, runs can be categorized by projects

results dictionary of results (read-only)

tag run tag (uid or workflow id if exists)

uid Unique run id

Methods

artifact_subpath(self, *subpaths)	subpaths under output path artifacts path
commit(self, message)	save run state and add a commit message
commit_children(self[, best_run])	update/commit all children, and optionally mark the best note: best_run marks the child iteration number (starts from 1)
from_dict(attrs[, rundb, autocommit, tmp, ...])	create execution context from dict
get_child(self, **params)	get child context (iteration)
get_dataitem(self, url)	get mlrun dataitem from url
get_input(self, key, url)	get an input data object, data objects have methods such as .get(), .download(), .url, ..
get_meta(self)	Reserved for internal use
get_param(self, key[, default])	get a run parameter, or use the provided default if not set
get_secret(self, key)	get a key based secret e.g.
log_artifact(self, item[, body, local_path, ...])	log an output artifact and optionally upload it to data-store
log_dataset(self, key, df[, tag, ...])	log a dataset artifact and optionally upload it to data-store
log_iteration_results(self, best, summary, task)	Reserved for internal use
log_metric(self, key, value[, timestamp, labels])	TBD, log a real-time time-series metric
log_metrics(self, keyvals[, timestamp, labels])	TBD, log a set of real-time time-series metrics
log_model(self, key[, body, framework, tag, ...])	log a model artifact and optionally upload it to data-store
log_result(self, key, value[, commit])	log a scalar result value
log_results(self, results[, commit])	log a set of scalar result values
set_annotation(self, key, value, replace)	set/record a specific annotation
set_hostname(self, host)	update the hostname
set_label(self, key, value, replace)	set/record a specific label
set_state(self, state, error[, commit])	modify and store the run state or mark an error
to_dict(self)	convert the run context to a dictionary
to_json(self)	convert the run context to a json buffer
to_yaml(self)	convert the run context to a yaml buffer

set_logger_stream	

property annotations
dictionary with annotations (read-only)

artifact_subpath(*self*, **subpaths*)
subpaths under output path artifacts path

property artifacts
dictionary of artifacts (read-only)

commit(*self*, *message: str = ''*)
save run state and add a commit message

commit_children(*self*, *best_run=0*)
update/commit all children, and optionally mark the best note: best_run marks the child iteration number (starts from 1)

classmethod from_dict(*attrs: dict*, *rundb=''*, *autocommit=False*, *tmp=''*, *host=None*, *log_stream=None*)
create execution context from dict

get_child(*self*, ***params*)
get child context (iteration)

allow sub experiments (epochs, hyper-param, ..) under a parent will create a new iteration, log_xx will update the child only use commit_children() to save all the children and specify the best run

Parameters ****params** – extra (or override) params to parent context

get_dataitem(*self*, *url*)
get mlrun dataitem from url

get_input(*self*, *key: str*, *url: str = ''*)
get an input data object, data objects have methods such as .get(), .download(), .url, .. to access the actual data

get_meta(*self*)
Reserved for internal use

get_param(*self*, *key: str*, *default=None*)
get a run parameter, or use the provided default if not set

get_secret(*self*, *key: str*)
get a key based secret e.g. DB password from the context secrets can be specified when invoking a run through files, env, ..

property in_path
default input path for data objects

property inputs
dictionary of input data items (read-only)

property iteration
child iteration index, for hyper parameters

kind = 'run'

property labels
dictionary with labels (read-only)

log_artifact(*self*, *item*, *body=None*, *local_path=None*, *artifact_path=None*, *tag=''*, *viewer=None*, *target_path=''*, *src_path=None*, *upload=None*, *labels=None*, *format=None*, *db_key=None*, ***kwargs*)
log an output artifact and optionally upload it to datastore

Parameters

- **item** – artifact key or artifact class ()

- **body** – will use the body as the artifact content
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use:

  artifact_path=context.artifact_subpath('data')
- **tag** – version tag
- **viewer** – kubeflow viewer type
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **src_path** – deprecated, use local_path
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_dataset(*self*, *key*, *df*, *tag=''*, *local_path=None*, *artifact_path=None*, *upload=True*, *labels=None*, *format=''*, *preview=None*, *stats=False*, *db_key=None*, *target_path=''*, *extra_data=None*, ***kwargs*)

log a dataset artifact and optionally upload it to datastore

Parameters

- **key** – artifact key
- **df** – dataframe object
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use:

  artifact_path=context.artifact_subpath('data')
- **tag** – version tag
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **preview** – number of lines to store as preview in the artifact metadata
- **stats** – calculate and store dataset stats in the artifact metadata
- **extra_data** – key/value list of extra files/charts to link with this dataset
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_iteration_results(*self*, *best*, *summary: list*, *task: dict*, *commit=False*)

Reserved for internal use

property log_level
get the logging level, e.g. 'debug', 'info', 'error'

log_metric(*self*, *key: str*, *value*, *timestamp=None*, *labels=None*)
TBD, log a real-time time-series metric

log_metrics(*self*, *keyvals: dict*, *timestamp=None*, *labels=None*)
TBD, log a set of real-time time-series metrics

log_model(*self*, *key*, *body=None*, *framework=''*, *tag=''*, *model_dir=None*, *model_file=None*, *metrics=None*, *parameters=None*, *artifact_path=None*, *upload=True*, *labels=None*, *inputs=None*, *outputs=None*, *extra_data=None*, *db_key=None*)
log a model artifact and optionally upload it to datastore

Parameters

- **key** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **model_file** – path to the local model file we upload (seel also model_dir)
- **model_dir** – path to the local dir holding the model file and extra files
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use:

  artifact_path=context.artifact_subpath('data')
- **framework** – name of the ML framework
- **tag** – version tag
- **metrics** – key/value dict of model metrics
- **parameters** – key/value dict of model parameters
- **inputs** – ordered list of model input features (name, type, ..)
- **outputs** – ordered list of model output/result elements (name, type, ..)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **extra_data** – key/value list of extra files/charts to link with this dataset value can be abs/relative path string | bytes | artifact object
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_result(*self*, *key: str*, *value*, *commit=False*)
log a scalar result value

Parameters

- **key** – result key
- **value** – result value
- **commit** – commit (write to DB now vs wait for the end of the run)

log_results(*self*, *results: dict*, *commit=False*)
log a set of scalar result values

Parameters

- **results** – key/value dict or results
- **commit** – commit (write to DB now vs wait for the end of the run)

property logger
built-in logger interface

property out_path
default output path for artifacts

property parameters
dictionary of run parameters (read-only)

property project
project name, runs can be categorized by projects

property results
dictionary of results (read-only)

set_annotation(*self*, *key: str*, *value*, *replace: bool = True*)
set/record a specific annotation

set_hostname(*self*, *host: str*)
update the hostname

set_label(*self*, *key: str*, *value*, *replace: bool = True*)
set/record a specific label

set_logger_stream(*self*, *stream*)

set_state(*self*, *state: str = None*, *error: str = None*, *commit=True*)
modify and store the run state or mark an error

Parameters

- **state** – set run state
- **error** – error message (if exist will set the state to error)
- **commit** – will immediately update the state in the DB

property tag
run tag (uid or workflow id if exists)

to_dict(*self*)
convert the run context to a dictionary

to_json(*self*)
convert the run context to a json buffer

to_yaml(*self*)
convert the run context to a yaml buffer

property uid
Unique run id

exception mlrun.execution.**MLCtxValueError**
Bases: Exception

1.12.5 mlrun.model

class mlrun.model.**BaseMetadata**(*name=None*, *tag=None*, *hash=None*, *namespace=None*, *project=None*, *labels=None*, *annotations=None*, *categories=None*, *updated=None*)

Bases: *mlrun.model.ModelObj*

Methods

copy
from_dict
to_dict
to_json
to_str
to_yaml

class mlrun.model.**ImageBuilder**(*functionSourceCode=None*, *source=None*, *image=None*, *base_image=None*, *commands=None*, *secret=None*, *code_origin=None*, *registry=None*)

Bases: *mlrun.model.ModelObj*

An Image builder

Methods

copy
from_dict
to_dict
to_json
to_str
to_yaml

base_image = None
base_image

codeEntryType = None
codeEntryType

code_origin = None
code_origin

commands = None
commands

functionSourceCode = None
functionSourceCode

image = None
image

registry = None
registry

secret = None
secret

source = None
course

class mlrun.model.**ModelObj**
Bases: object

Methods

copy	
from_dict	
to_dict	
to_json	
to_str	
to_yaml	

copy(*self*)

classmethod from_dict(*struct=None*, *fields=None*)

to_dict(*self*, *fields=None*, *exclude=None*)

to_json(*self*)

to_str(*self*)

to_yaml(*self*)

mlrun.model.**NewTask**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *tuning_strategy=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*)
Creates a new task - see new_task

class mlrun.model.**ObjectDict**(*classes_map*, *default_kind=''*)
Bases: object

Methods

copy	
from_dict	
items	
keys	
to_dict	
to_json	
to_str	
to_yaml	
values	

copy(*self*)

classmethod from_dict(*classes_map: dict*, *children=None*, *default_kind=''*)

items(*self*)

keys(*self*)

to_dict(*self*)

to_json(*self*)

to_str(*self*)

to_yaml(*self*)

values(*self*)

class mlrun.model.**RunMetadata**(*uid=None*, *name=None*, *project=None*, *labels=None*, *annotations=None*, *iteration=None*)

Bases: *mlrun.model.ModelObj*

Run metadata

Attributes

iteration

Methods

copy	
from_dict	
to_dict	
to_json	
to_str	
to_yaml	

property iteration

class mlrun.model.**RunObject**(*spec: mlrun.model.RunSpec = None*, *metadata: mlrun.model.RunMetadata = None*, *status: mlrun.model.RunStatus = None*)

Bases: *mlrun.model.RunTemplate*

A run

Attributes

metadata

outputs

spec

status

Methods

with_secrets(self, kind, source)	register a secrets source (file, env or dict)

artifact	
copy	
from_dict	
from_template	
logs	
output	
set_label	
show	
state	
to_dict	
to_env	
to_json	
to_str	
to_yaml	
uid	
with_hyper_params	
with_input	
with_param_file	
with_params	

artifact(*self*, *key*)

classmethod from_template(*template: mlrun.model.RunTemplate*)

logs(*self*, *watch=True*, *db=None*)

output(*self*, *key*)

property outputs

show(*self*)

state(*self*)

property status

uid(*self*)

class mlrun.model.**RunSpec**(*parameters=None*, *hyperparams=None*, *param_file=None*, *selector=None*, *handler=None*, *inputs=None*, *outputs=None*, *input_path=None*, *output_path=None*, *function=None*, *secret_sources=None*, *data_stores=None*, *tuning_strategy=None*, *verbose=None*, *scrape_metrics=False*)

Bases: *mlrun.model.ModelObj*

Run specification

Attributes

data_stores

handler_name

inputs

outputs

secret_sources

Methods

copy	
from_dict	
to_dict	
to_json	
to_str	
to_yaml	

property data_stores

property handler_name

property inputs

property outputs

property secret_sources

to_dict(*self*, *fields=None*, *exclude=None*)

class mlrun.model.**RunStatus**(*state=None*, *error=None*, *host=None*, *commit=None*, *status_text=None*, *results=None*, *artifacts=None*, *start_time=None*, *last_update=None*, *iterations=None*)

Bases: *mlrun.model.ModelObj*

Run status

Methods

copy	
from_dict	
to_dict	
to_json	
to_str	
to_yaml	

class mlrun.model.**RunTemplate**(*spec: mlrun.model.RunSpec = None*, *metadata: mlrun.model.RunMetadata = None*)

Bases: *mlrun.model.ModelObj*

Run template

Attributes

metadata

spec

Methods

with_secrets(self, kind, source)	register a secrets source (file, env or dict)

copy	
from_dict	
set_label	
to_dict	
to_env	
to_json	
to_str	
to_yaml	
with_hyper_params	
with_input	
with_param_file	
with_params	

property metadata

set_label(*self*, *key*, *value*)

property spec

to_env(*self*)

with_hyper_params(*self*, *hyperparams*, *selector=None*, *strategy=None*)

with_input(*self*, *key*, *path*)

with_param_file(*self*, *param_file*, *selector=None*, *strategy=None*)

with_params(*self*, ***kwargs*)

with_secrets(*self*, *kind*, *source*)

register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, e.g.

proj.with_secrets('file', 'file.txt') proj.with_secrets('inline', {'key': 'val'}) proj.with_secrets('env', 'ENV1,ENV2')

Parameters

- **kind** – secret type (file, inline, env)
- **source** – secret data or link (see example)

Returns project object

mlrun.model.**new_task**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *tuning_strategy=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*)

Creates a new task

Parameters

- **name** – task name
- **project** – task project

:param handler code entry-point/hanfler name :param params: input parameters (dict) :param hyper_params: dictionary of hyper parameters and list values, each

hyper param holds a list of values, the run will be executed for every parameter combination (GridSearch)

Parameters

- **param_file** – a csv file with parameter combinations, first row hold the parameter names, following rows hold param values
- **selector** – selection criteria for hyper params e.g. "max.accuracy"
- **tuning_strategy** – selection strategy for hyper params e.g. list, grid, random
- **inputs** – dictionary of input objects + optional paths (if path is omitted the path will be the in_path/key.
- **outputs** – dictionary of input objects + optional paths (if path is omitted the path will be the out_path/key.
- **in_path** – default input path/url (prefix) for inputs
- **out_path** – default output path/url (prefix) for artifacts
- **artifact_path** – default artifact output path
- **secrets** – extra secrets specs, will be injected into the runtime e.g. ['file=<filename>', 'env=ENV_KEY1,ENV_KEY2']
- **base** – task instance to use as a base instead of a fresh new task instance

1.12.6 mlrun.platforms

mlrun.platforms.**auto_mount**(*pvc_name=''*, *volume_mount_path=''*, *volume_name=None*)
choose the mount based on env variables and params

volume will be selected by the following order: - k8s PVC volume when both pvc_name and volume_mount_path are set - iguazio v3io volume when V3IO_ACCESS_KEY and V3IO_USERNAME env vars are set - k8s PVC volume when env var is set: MLRUN_PVC_MOUNT=<pvc-name>:<mount-path>

mlrun.platforms.**mount_pvc**(*pvc_name*, *volume_name='pipeline'*, *volume_mount_path='/mnt/pipeline'*)
Modifier function to apply to a Container Op to simplify volume, volume mount addition and enable better reuse of volumes, volume claims across container ops. Usage:

train = train_op(...) train.apply(mount_pvc('claim-name', 'pipeline', '/mnt/pipeline'))

mlrun.platforms.**mount_v3io**(*name='v3io'*, *remote='~/'*, *mount_path='/User'*, *access_key=''*, *user=''*, *secret=None*)
Modifier function to apply to a Container Op to volume mount a v3io path

Parameters

- **name** – the volume name
- **remote** – the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/
- **mount_path** – the volume mount path
- **access_key** – the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used

- **user** – the username used to auth against v3io. if not given V3IO_USERNAME env var will be used
- **secret** – k8s secret name which would be used to get the username and access key to auth against v3io.

mlrun.platforms.**v3io_cred**(*api=''*, *user=''*, *access_key=''*)
Modifier function to copy local v3io env vars to task Usage:

train = train_op(. . .) train.apply(use_v3io_cred())

1.12.7 mlrun.projects

mlrun.projects.**load_project**(*context*, *url=None*, *name=None*, *secrets=None*, *init_git=False*, *subpath=''*, *clone=False*)
Load an MLRun project from git or tar or dir

Parameters

- **context** – project local directory path
- **url** – git or tar.gz sources archive path e.g.: git://github.com/mlrun/demo-xgb-project.git
- **name** – project name
- **secrets** – key:secret dict or SecretsStore used to download sources
- **init_git** – if True, will git init the context dir
- **subpath** – project subpath (within the archive)
- **clone** – if True, always clone (delete any existing content)

Returns project object

mlrun.projects.**new_project**(*name*, *context=None*, *init_git=False*)
Create a new MLRun project

Parameters

- **name** – project name
- **context** – project local directory path
- **init_git** – if True, will git init the context dir

Returns project object

class mlrun.projects.**MlrunProject**(*name=None*, *description=None*, *params=None*, *functions=None*, *workflows=None*, *artifacts=None*, *artifact_path=None*, *conda=None*)
Bases: *mlrun.model.ModelObj*

Attributes

artifacts list of artifacts used in this project

functions list of function object/specs used in this project

mountdir specify to mount the context dir inside the function container

source source url or git repo

workflows list of workflows specs used in this project

Methods

clear_context(self)	delete all files and clear the context dir
create_remote(self, url[, name])	create remote for the project git
func(self, key[, sync])	get function object by name
get_param(self, key[, default])	get project param by key
get_secret(self, key)	get a key based secret e.g.
pull(self[, branch, remote])	pull/update sources from git or tar into the context dir
push(self, branch[, message, update, remote])	update spec and push updates to remote git repo
register_artifacts(self)	register the artifacts in the MLRun DB (under this project)
reload(self[, sync])	reload the project and function objects from yaml/specs
run(self[, name, workflow_path, arguments, ...])	run a workflow using kubeflow pipelines
save(self[, filepath])	save the project object into a file (default to project.yaml)
save_workflow(self, name, target[, ...])	create and save a workflow as a yaml or archive file
set_function(self, func[, name, kind, ...])	update or add a function object to the project
set_workflow(self, name, workflow_path[, embed])	add or update a workflow, specify a name and the code path
sync_functions(self, names[, always, save])	reload function objects from specs and files
with_secrets(self, kind, source[, prefix])	register a secrets source (file, env or dict)

copy	
from_dict	
get_run_status	
log_artifact	
to_dict	
to_json	
to_str	
to_yaml	

property artifacts
list of artifacts used in this project

clear_context (*self*)
delete all files and clear the context dir

create_remote (*self*, *url*, *name='origin'*)
create remote for the project git

Parameters

- **url** – remote git url
- **name** – name for the remote (default is 'origin')

func (*self*, *key*, *sync=False*)
get function object by name

Parameters **sync** – will reload/reinit the function

Returns function object

property functions
list of function object/specs used in this project

get_param(*self*, *key: str*, *default=None*)
get project param by key

get_run_status(*self*, *workflow_id*, *timeout=3600*, *expected_statuses=None*, *notifiers: mlrun.utils.helpers.RunNotifications = None*)

get_secret(*self*, *key: str*)
get a key based secret e.g. DB password from the context secrets can be specified when invoking a run through files, env, ..

kind = 'project'

log_artifact(*self*, *item*, *body=None*, *tag=''*, *local_path=''*, *artifact_path=None*, *format=None*, *upload=True*, *labels=None*, *target_path=None*)

property mountdir
specify to mount the context dir inside the function container use '.' to use the same path as in the client e.g. Jupyter

pull(*self*, *branch=None*, *remote=None*)
pull/update sources from git or tar into the context dir

Parameters

- **branch** – git branch, if not the current one
- **remote** – git remote, if other than origin

push(*self*, *branch*, *message=None*, *update=True*, *remote=None*, *add: list = None*)
update spec and push updates to remote git repo

Parameters

- **branch** – target git branch
- **message** – git commit message
- **update** – update files (git add update=True)
- **remote** – git remote, default to origin
- **add** – list of files to add

register_artifacts(*self*)
register the artifacts in the MLRun DB (under this project)

reload(*self*, *sync=False*)
reload the project and function objects from yaml/specs

Parameters **sync** – set to True to load functions objects

Returns project object

run(*self*, *name=None*, *workflow_path=None*, *arguments=None*, *artifact_path=None*, *namespace=None*, *sync=False*, *watch=False*, *dirty=False*, *ttl=None*)
run a workflow using kubeflow pipelines

Parameters

- **name** – name of the workflow
- **workflow_path** – url to a workflow file, if not a project workflow
- **arguments** – kubeflow pipelines arguments (parameters)

- **artifact_path** – target path/url for workflow artifacts, the string '{{workflow.uid}}' will be replaced by workflow id
- **namespace** – kubernetes namespace if other than default
- **sync** – force functions sync before run
- **watch** – wait for pipeline completion
- **dirty** – allow running the workflow when the git repo is dirty

:param ttl pipeline ttl in secs (after that the pods will be removed)

Returns run id

save(*self*, *filepath=None*)
save the project object into a file (default to project.yaml)

save_workflow(*self*, *name*, *target*, *artifact_path=None*, *ttl=None*)
create and save a workflow as a yaml or archive file

Parameters

- **name** – workflow name
- **target** – target file path (can end with .yaml or .zip)
- **artifact_path** – target path/url for workflow artifacts, the string '{{workflow.uid}}' will be replaced by workflow id

:param ttl pipeline ttl in secs (after that the pods will be removed)

set_function(*self*, *func*, *name=''*, *kind=''*, *image=None*, *with_repo=None*)
update or add a function object to the project

function can be provided as an object (func) or a .py/.ipynb/.yaml url support url prefixes:

object (s3://, v3io://, ..) MLRun DB e.g. db://project/func:ver functions hub/market: e.g. hub://sklearn_classifier:master

examples:

proj.set_function(func_object) proj.set_function('./src/mycode.py', 'ingest',

image='myrepo/ing:latest', with_repo=True)

proj.set_function('http://.../mynb.ipynb', 'train') proj.set_function('./func.yaml') proj.set_function('hub://get_toy_data', 'getdata')

Parameters

- **func** – function object or spec/code url
- **name** – name of the function (under the project)
- **kind** – runtime kind e.g. job, nuclio, spark, dask, mpijob default: job
- **image** – docker image to be used, can also be specified in the function object/yaml
- **with_repo** – add (clone) the current repo to the build source

Returns project object

set_workflow(*self*, *name*, *workflow_path: str*, *embed=False*, ***args*)
add or update a workflow, specify a name and the code path

property source
source url or git repo

sync_functions (*self*, *names: list = None*, *always=True*, *save=False*)
reload function objects from specs and files

with_secrets (*self*, *kind*, *source*, *prefix=''*)
register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, e.g.

proj.with_secrets('file', 'file.txt') proj.with_secrets('inline', {'key': 'val'}) proj.with_secrets('env', 'ENV1,ENV2', prefix='PFX_')

Parameters

- **kind** – secret type (file, inline, env)
- **source** – secret data or link (see example)
- **prefix** – add a prefix to the keys in this source

Returns project object

property workflows
list of workflows specs used in this project

1.12.8 mlrun.run

class mlrun.run.**RunStatuses**
Bases: object

Methods

all	
stable_statuses	
transient_statuses	

static all()

error = 'Error'

failed = 'Failed'

running = 'Running'

skipped = 'Skipped'

static stable_statuses()

succeeded = 'Succeeded'

static transient_statuses()

mlrun.run.**code_to_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *filename: str = ''*, *handler: str = ''*, *kind: str = ''*, *image: str = None*, *code_output=''*, *embed_code=True*, *description=''*, *categories: list = None*, *labels: dict = None*, *with_doc=True*)
convert code or notebook to function object with embedded code code stored in the function spec and can be refreshed using .with_code() eliminate the need to build container images every time we edit the code

Parameters

- **name** – function name

- **project** – function project (none for 'default')
- **tag** – function tag (none for 'latest')
- **filename** – blank for current notebook, or path to .py/.ipynb file
- **handler** – name of function handler (if not main)
- **kind** – optional, runtime type local, job, dask, mpijob, ..
- **image** – optional, container image
- **code_output** – save the generated code (from notebook) in that path
- **embed_code** – embed the source code into the function spec
- **description** – function description
- **categories** – list of categories (for function marketplace)
- **labels** – dict of label names and values to tag the function
- **with_doc** – document the function parameters

Returns function object

mlrun.run.**download_object**(*url*, *target*, *secrets=None*)
download mlrun dataitem (from path/url to target path)

mlrun.run.**function_to_module**(*code=''*, *workdir=None*, *secrets=None*)
Load code, notebook or mlrun function as .py module this function can import a local/remote py file or notebook or load an mlrun function object as a module, you can use this from your code, notebook, or another function (for common libs)

Note: the function may have package requirements which must be satisfied

example:

mod = mlrun.function_to_module('./examples/training.py') task = mlrun.new_task(inputs={'infile.txt': '../examples/infile.txt'}) context = mlrun.get_or_create_ctx('myfunc', spec=task) mod.my_job(context, p1=1, p2='x') print(context.to_yaml())

fn = mlrun.import_function('hub://open_archive') mod = mlrun.function_to_module(fn) data = mlrun.run.get_dataitem("https://fpsignals-public.s3.amazonaws.com/catsndogs.tar.gz") context = mlrun.get_or_create_ctx('myfunc') mod.open_archive(context, archive_url=data) print(context.to_yaml())

Parameters

- **code** – path/url to function (.py or .ipynb or .yaml) OR function object
- **workdir** – code workdir
- **secrets** – secrets needed to access the URL (e.g.s3, v3io, ..)

Returns python module

mlrun.run.**get_dataitem**(*url*, *secrets=None*, *db=None*)
get mlrun dataitem object (from path/url)

mlrun.run.**get_object**(*url*, *secrets=None*, *size=None*, *offset=0*, *db=None*)
get mlrun dataitem body (from path/url)

mlrun.run.**get_or_create_ctx**(*name: str*, *event=None*, *spec=None*, *with_env: bool = True*, *rundb: str = ''*)

called from within the user program to obtain a run context

the run context is an interface for receiving parameters, data and logging run results, the run context is read from the event, spec, or environment (in that order), user can also work without a context (local defaults mode)

all results are automatically stored in the "rundb" or artifact store, the path to the rundb can be specified in the call or obtained from env.

Parameters

- **name** – run name (will be overridden by context)
- **event** – function (nuclio Event object)
- **spec** – dictionary holding run spec
- **with_env** – look for context in environment vars, default True
- **rundb** – path/url to the metadata and artifact database

Returns execution context

Example:

load MLRUN runtime context (will be set by the runtime framework e.g. KubeFlow) context = get_or_create_ctx('train')

get parameters from the runtime context (or use defaults) p1 = context.get_param('p1', 1) p2 = context.get_param('p2', 'a-string')

access input metadata, values, files, and secrets (passwords) print(f'Run: {context.name} (uid={context.uid})') print(f'Params: p1={p1}, p2={p2}') print('accesskey = {}'.format(context.get_secret('ACCESS_KEY'))) print('file: {}'.format(context.get_input('infile.txt').get()))

RUN some useful code e.g. ML training, data prep, etc.

log scalar result values (job result metrics) context.log_result('accuracy', p1 * 2) context.log_result('loss', p1 * 3) context.set_label('framework', 'sklearn')

log various types of artifacts (file, web page, table), will be versioned and visible in the UI context.log_artifact('model.txt', body=b'abc is 123', labels={'framework': 'xgboost'}) context.log_artifact('results.html', body=b'<b> Some HTML <b>', viewer='web-app')

mlrun.run.**get_pipeline**(*run_id*, *namespace=None*)

Get Pipeline status

Parameters

- **run_id** – id of pipelines run
- **namespace** – k8s namespace if not default

Returns kfp run dict

mlrun.run.**import_function**(*url=''*, *secrets=None*, *db=''*)

Create function object from DB or local/remote YAML file

Reading from a file or remote URL (http(s), s3, git, v3io, ..)

Parameters

- **url** – path/url to function YAML file or 'db://{project}/{name}[:tag]' when reading from mlrun db
- **secrets** – optional, credentials dict for DB or URL (s3, v3io, ...)

- **db** – optional, mlrun api/db path

Returns function object

mlrun.run.**import_function_to_dict**(*url*, *secrets=None*)
Load function spec from local/remote YAML file

mlrun.run.**list_pipelines**(*full=False*, *page_token=''*, *page_size=10*, *sort_by=''*, *experiment_id=None*, *namespace=None*)
List pipelines

mlrun.run.**new_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *kind: str = ''*, *command: str = ''*, *image: str = ''*, *args: list = None*, *runtime=None*, *mode=None*, *kfp=None*)
Create a new ML function from base properties

e.g.: # define a container based function f = new_function(command='job://training.py -v', image='myrepo/image:latest')

define a handler function (execute a local function handler) f = new_function().run(task, handler=myfunction)

Parameters

- **name** – function name
- **project** – function project (none for 'default')
- **tag** – function version tag (none for 'latest')
- **kind** – runtime type (local, job, nuclio, spark, mpijob, dask, ..)
- **command** – command/url + args (e.g.: training.py –verbose)
- **image** – container image (start with '.' for default registry)
- **args** – command line arguments (override the ones in command)
- **runtime** – runtime (job, nuclio, spark, dask ..) object/dict store runtime specific details and preferences
- **mode** – runtime mode, e.g. noctx, pass to bypass mlrun
- **kfp** – reserved, flag indicating running within kubeflow pipeline

Returns function object

mlrun.run.**parse_command**(*runtime*, *url*)

mlrun.run.**run_local**(*task=None*, *command=''*, *name: str = ''*, *args: list = None*, *workdir=None*, *project: str = ''*, *tag: str = ''*, *secrets=None*, *handler=None*, *params: dict = None*, *inputs: dict = None*, *artifact_path: str = ''*)
Run a task on function/code (.py, .ipynb or .yaml) locally,

e.g.: # define a task task = new_task(params={'p1': 8}, out_path=out_path) # run run = run_local(spec, command='src/training.py', workdir='src')

or specify base task parameters (handler, params, ..) in the call

run = run_local(handler=my_function, params={'x': 5})

Parameters

- **task** – task template object or dict (see RunTemplate)
- **command** – command/url/function

- **name** – ad hook function name
- **args** – command line arguments (override the ones in command)
- **workdir** – working dir to exec in
- **project** – function project (none for 'default')
- **tag** – function version tag (none for 'latest')
- **secrets** – secrets dict if the function source is remote (s3, v3io, ..)
- **handler** – pointer or name of a function handler
- **params** – input parameters (dict)
- **inputs** – input objects (dict of key: path)
- **artifact_path** – default artifact output path

Returns run object

mlrun.run.**run_pipeline**(*pipeline*, *arguments=None*, *project=None*, *experiment=None*, *run=None*, *namespace=None*, *artifact_path=None*, *ops=None*, *url=None*, *ttl=None*)

remote KubeFlow pipeline execution

Submit a workflow task to KFP via mlrun API service

:param pipeline KFP pipeline function or path to .yaml/.zip pipeline file :param arguments pipeline arguments :param experiment experiment name :param run optional, run name :param namespace Kubernetes namespace (if not using default) :param url optional, url to mlrun API service :param artifact_path target location/url for mlrun artifacts :param ops additional operators (.apply() to all pipeline functions) :param ttl pipeline ttl in secs (after that the pods will be removed)

Returns kubeflow pipeline id

mlrun.run.**wait_for_pipeline_completion**(*run_id*, *timeout=3600*, *expected_statuses: List[str] = None*, *namespace=None*)

Wait for Pipeline status, timeout in sec

Parameters

- **run_id** – id of pipelines run
- **timeout** – wait timeout in sec
- **expected_statuses** – list of expected statuses, one of [Succeeded | Failed | Skipped | Error], by default [Succeeded]
- **namespace** – k8s namespace if not default

Returns kfp run dict

1.12.9 mlrun.utils

class mlrun.utils.**Enum**

Generic enumeration.

Derive from this class to define new enumerations.

name

The name of the Enum member.

value

The value of the Enum member.

class mlrun.utils.**FormatterKinds**
An enumeration.

class mlrun.utils.**HumanReadableFormatter**

Methods

converter([seconds])	Convert seconds since the Epoch to a time tuple expressing local time.
format(self, record)	Format the specified record as text.
formatException(self, ei)	Format and return the specified exception information as a string.
formatStack(self, stack_info)	This method is provided as an extension point for specialized formatting of stack information.
formatTime(self, record[, datefmt])	Return the creation time of the specified LogRecord as formatted text.
usesTime(self)	Check if the format uses the creation time of the record.

formatMessage	

format(*self*, *record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class mlrun.utils.**IO**
Generic base class for TextIO and BinaryIO.

This is an abstract, generic version of the return of open().

NOTE: This does not distinguish between the different possible classes (text vs. binary, read vs. write vs. read/write, append-only, unbuffered). The TextIO and BinaryIO subclasses below capture the distinctions between text vs. binary, which is pervasive in the interface; however we currently do not offer a way to track the other distinctions in the type system.

Attributes

closed

mode

name

Methods

close	
fileno	
flush	
isatty	
read	
readable	
readline	
readlines	
seek	
seekable	
tell	
truncate	
writable	
write	
writelines	

class mlrun.utils.**JSONFormatter**

Methods

converter([seconds])	Convert seconds since the Epoch to a time tuple expressing local time.
format(self, record)	Format the specified record as text.
formatException(self, ei)	Format and return the specified exception information as a string.
formatStack(self, stack_info)	This method is provided as an extension point for specialized formatting of stack information.
formatTime(self, record[, datefmt])	Return the creation time of the specified LogRecord as formatted text.
usesTime(self)	Check if the format uses the creation time of the record.

formatMessage	

format(*self*, *record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class mlrun.utils.**MyEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Methods

default(self, obj)	Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).
encode(self, o)	Return a JSON string representation of a Python data structure.
iterencode(self, o[, _one_shot])	Encode the given object and yield each string representation as available.

default (*self*, *obj*)

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

exception mlrun.utils.**RepresenterError**

class mlrun.utils.**Timestamp**

Pandas replacement for python datetime.datetime object.

Timestamp is the pandas equivalent of python's Datetime and is interchangeable with it in most cases. It's the type used for the entries that make up a DatetimeIndex, and other timeseries oriented data structures in pandas.

Parameters

ts_input [datetime-like, str, int, float] Value to be converted to Timestamp.

freq [str, DateOffset] Offset which Timestamp will have.

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will have.

unit [str] Unit used for conversion if ts_input is of type int or float. The valid values are 'D', 'h', 'm', 's', 'ms', 'us', and 'ns'. For example, 's' means seconds and 'ms' means milliseconds.

year, month, day [int]

hour, minute, second, microsecond [int, optional, default 0]

nanosecond [int, optional, default 0]

tzinfo [datetime.tzinfo, optional, default None]

fold [{0, 1}, default None, keyword-only] Due to daylight saving time, one wall clock time can occur twice when shifting from summer to winter time; fold describes whether the datetime-like corresponds to the first (0) or the second time (1) the wall clock hits the ambiguous time

New in version 1.1.0.

Notes

There are essentially three calling conventions for the constructor. The primary form accepts four parameters. They can be passed by position or keyword.

The other two forms mimic the parameters from `datetime.datetime`. They can be passed by either position or keyword, but not both mixed together.

Examples

Using the primary calling convention:

This converts a datetime-like string

```
>>> pd.Timestamp('2017-01-01T12')
Timestamp('2017-01-01 12:00:00')
```

This converts a float representing a Unix epoch in units of seconds

```
>>> pd.Timestamp(1513393355.5, unit='s')
Timestamp('2017-12-16 03:02:35.500000')
```

This converts an int representing a Unix-epoch in units of seconds and for a particular timezone

```
>>> pd.Timestamp(1513393355, unit='s', tz='US/Pacific')
Timestamp('2017-12-15 19:02:35-0800', tz='US/Pacific')
```

Using the other two forms that mimic the API for `datetime.datetime`:

```
>>> pd.Timestamp(2017, 1, 1, 12)
Timestamp('2017-01-01 12:00:00')
```

```
>>> pd.Timestamp(year=2017, month=1, day=1, hour=12)
Timestamp('2017-01-01 12:00:00')
```

Attributes

asm8 Return numpy datetime64 format in nanoseconds.

day

day_of_week Return day of the week.

day_of_year Return the day of the year.

dayofweek Return day of the week.

dayofyear Return the day of the year.

days_in_month Return the number of days in the month.

daysinmonth Return the number of days in the month.

fold

freq

freqstr Return the total number of days in the month.

hour

is_leap_year Return True if year is a leap year.

is_month_end Return True if date is last day of month.

is_month_start Return True if date is first day of month.

is_quarter_end Return True if date is last day of the quarter.

is_quarter_start Return True if date is first day of the quarter.

is_year_end Return True if date is last day of the year.

is_year_start Return True if date is first day of the year.

microsecond

minute

month

nanosecond

quarter Return the quarter of the year.

second

tz Alias for tzinfo.

tzinfo

value

week Return the week number of the year.

weekofyear Return the week number of the year.

year

Methods

astimezone(self, tz)	Convert tz-aware Timestamp to another time zone.
ceil(self, freq[, ambiguous, nonexistent])	Return a new Timestamp ceiled to this resolution.
combine(date, time)	Combine date, time into datetime with same date and time fields.
ctime()	Return ctime() style string.
date()	Return date object with same year, month and day.
day_name()	Return the day name of the Timestamp with specified locale.
dst()	Return self.tzinfo.dst(self).
floor(self, freq[, ambiguous, nonexistent])	Return a new Timestamp floored to this resolution.
fromisoformat()	string -> datetime from datetime.isoformat() output
fromordinal(ordinal[, freq, tz])	Passed an ordinal, translate and convert to a ts.
fromtimestamp(ts)	Transform timestamp[, tz] to tz's local time from POSIX timestamp.
isocalendar()	Return a 3-tuple containing ISO year, week number, and weekday.
isoformat()	[sep] -> string in ISO 8601 format, YYYY-MM-DDT[HH[:MM[:SS[.mmm[uuu]]]]][+HH:MM].
isoweekday()	Return the day of the week represented by the date.
month_name()	Return the month name of the Timestamp with specified locale.

Continued on next page

Table 12 – continued from previous page

normalize()	Normalize Timestamp to midnight, preserving tz information.
now([tz])	Return new Timestamp object representing current time local to tz.
replace(self[, year, month, day, hour, ...])	Implements datetime.replace, handles nanoseconds.
round(self, freq[, ambiguous, nonexistent])	Round the Timestamp to the specified resolution.
strftime(format)	Return a string representing the given POSIX timestamp controlled by an explicit format string.
strptime(string, format)	Function is not implemented.
time()	Return time object with same time but with tzinfo=None.
timestamp()	Return POSIX timestamp as float.
timetuple()	Return time tuple, compatible with time.localtime().
timetz()	Return time object with same time and tzinfo.
to_datetime64()	Return a numpy.datetime64 object with 'ns' precision.
to_julian_date(self)	Convert TimeStamp to a Julian Date.
to_numpy()	Convert the Timestamp to a NumPy datetime64.
to_period()	Return an period of which this timestamp is an observation.
to_pydatetime()	Convert a Timestamp object to a native Python datetime object.
today(cls[, tz])	Return the current time in the local timezone.
toordinal()	Return proleptic Gregorian ordinal.
tz_convert(self, tz)	Convert tz-aware Timestamp to another time zone.
tz_localize(self, tz[, ambiguous, nonexistent])	Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.
tzname()	Return self.tzinfo.tzname(self).
utcfromtimestamp(ts)	Construct a naive UTC datetime from a POSIX timestamp.
utcnow()	Return a new Timestamp representing UTC day and time.
utcoffset()	Return self.tzinfo.utcoffset(self).
utctimetuple()	Return UTC time tuple, compatible with time.localtime().
weekday()	Return the day of the week represented by the date.

astimezone(*self*, *tz*)

Convert tz-aware Timestamp to another time zone.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

Returns

converted [Timestamp]

Raises

TypeError If Timestamp is tz-naive.

ceil(*self*, *freq*, *ambiguous='raise'*, *nonexistent='raise'*)

Return a new Timestamp ceiled to this resolution.

Parameters

freq [str] Frequency string indicating the ceiling resolution.

ambiguous [bool or {'raise', 'NaT'}, default 'raise'] The behavior is as follows:

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
- 'NaT' will return NaT for an ambiguous time.
- 'raise' will raise an AmbiguousTimeError for an ambiguous time.

New in version 0.24.0.

nonexistent [{'raise', 'shift_forward', 'shift_backward, 'NaT', timedelta}, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time.
- 'shift_backward' will shift the nonexistent time backward to the closest existing time.
- 'NaT' will return NaT where there are nonexistent times.
- timedelta objects will shift nonexistent times by the timedelta.
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Raises

ValueError if the freq cannot be converted.

classmethod combine(*date*, *time*)
Combine date, time into datetime with same date and time fields.

daysinmonth
Return the number of days in the month.

floor(*self*, *freq*, *ambiguous='raise'*, *nonexistent='raise'*)
Return a new Timestamp floored to this resolution.

Parameters

freq [str] Frequency string indicating the flooring resolution.

ambiguous [bool or {'raise', 'NaT'}, default 'raise'] The behavior is as follows:

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
- 'NaT' will return NaT for an ambiguous time.
- 'raise' will raise an AmbiguousTimeError for an ambiguous time.

New in version 0.24.0.

nonexistent [{'raise', 'shift_forward', 'shift_backward, 'NaT', timedelta}, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time.
- 'shift_backward' will shift the nonexistent time backward to the closest existing time.
- 'NaT' will return NaT where there are nonexistent times.

- timedelta objects will shift nonexistent times by the timedelta.
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Raises

ValueError if the freq cannot be converted.

property freqstr
Return the total number of days in the month.

classmethod fromordinal(*ordinal*, *freq=None*, *tz=None*)
Passed an ordinal, translate and convert to a ts. Note: by definition there cannot be any tz info on the ordinal itself.

Parameters

ordinal [int] Date corresponding to a proleptic Gregorian ordinal.

freq [str, DateOffset] Offset to apply to the Timestamp.

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for the Timestamp.

classmethod fromtimestamp(*ts*)
Transform timestamp[, tz] to tz's local time from POSIX timestamp.

classmethod now(*tz=None*)
Return new Timestamp object representing current time local to tz.

Parameters

tz [str or timezone object, default None] Timezone to localize to.

replace(*self*, *year=None*, *month=None*, *day=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*, *nanosecond=None*, *tzinfo=<class 'object'>*, *fold=None*)
Implements datetime.replace, handles nanoseconds.

Parameters

year [int, optional]

month [int, optional]

day [int, optional]

hour [int, optional]

minute [int, optional]

second [int, optional]

microsecond [int, optional]

nanosecond [int, optional]

tzinfo [tz-convertible, optional]

fold [int, optional]

Returns

Timestamp with fields replaced

round(*self*, *freq*, *ambiguous='raise'*, *nonexistent='raise'*)
Round the Timestamp to the specified resolution.

Parameters

freq [str] Frequency string indicating the rounding resolution.

ambiguous [bool or {'raise', 'NaT'}, default 'raise'] The behavior is as follows:

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
- 'NaT' will return NaT for an ambiguous time.
- 'raise' will raise an AmbiguousTimeError for an ambiguous time.

New in version 0.24.0.

nonexistent [{'raise', 'shift_forward', 'shift_backward, 'NaT', timedelta}, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time.
- 'shift_backward' will shift the nonexistent time backward to the closest existing time.
- 'NaT' will return NaT where there are nonexistent times.
- timedelta objects will shift nonexistent times by the timedelta.
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Returns

a new Timestamp rounded to the given resolution of *freq*

Raises

ValueError if the freq cannot be converted

strftime(*format*)
Return a string representing the given POSIX timestamp controlled by an explicit format string.

Parameters

format [str] Format string to convert Timestamp to string. See strftime documentation for more information on the format string: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior.

classmethod strptime(*string*, *format*)
Function is not implemented. Use pd.to_datetime().

to_julian_date(*self*) → numpy.float64
Convert TimeStamp to a Julian Date. 0 Julian date is noon January 1, 4713 BC.

classmethod today(*cls*, *tz=None*)
Return the current time in the local timezone. This differs from datetime.today() in that it can be localized to a passed timezone.

Parameters

tz [str or timezone object, default None] Timezone to localize to.

property tz
Alias for tzinfo.

tz_convert(*self*, *tz*)
Convert tz-aware Timestamp to another time zone.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

Returns

converted [Timestamp]

Raises

TypeError If Timestamp is tz-naive.

tz_localize(*self*, *tz*, *ambiguous='raise'*, *nonexistent='raise'*)

Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding local time.

ambiguous [bool, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

The behavior is as follows:

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates).
- 'NaT' will return NaT for an ambiguous time.
- 'raise' will raise an AmbiguousTimeError for an ambiguous time.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

The behavior is as follows:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time.
- 'shift_backward' will shift the nonexistent time backward to the closest existing time.
- 'NaT' will return NaT where there are nonexistent times.
- timedelta objects will shift nonexistent times by the timedelta.
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Returns

localized [Timestamp]

Raises

TypeError If the Timestamp is tz-aware and tz is not None.

classmethod utcfromtimestamp(*ts*)

Construct a naive UTC datetime from a POSIX timestamp.

classmethod utcnow()

Return a new Timestamp representing UTC day and time.

weekofyear
Return the week number of the year.

mlrun.utils.**create_class**(*pkg_class: str*)
Create a class from a package.module.class string

Parameters **pkg_class** – full class location, e.g. "sklearn.model_selection.GroupKFold"

mlrun.utils.**create_function**(*pkg_func: list*)
Create a function from a package.module.function string

Parameters **pkg_func** – full function location, e.g. "sklearn.feature_selection.f_classif"

class mlrun.utils.**datetime**(*year*, *month*, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])
The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints.

Attributes

day

fold

hour

microsecond

minute

month

second

tzinfo

year

Methods

astimezone()	tz -> convert to local time in new timezone tz
combine()	date, time -> datetime with same date and time fields
ctime()	Return ctime() style string.
date()	Return date object with same year, month and day.
dst()	Return self.tzinfo.dst(self).
fromisoformat()	string -> datetime from datetime.isoformat() output
fromordinal()	int -> date corresponding to a proleptic Gregorian ordinal.
fromtimestamp()	timestamp[, tz] -> tz's local time from POSIX timestamp.
isocalendar()	Return a 3-tuple containing ISO year, week number, and weekday.
isoformat()	[sep] -> string in ISO 8601 format, YYYY-MM-DDT[HH[:MM[:SS[.mmm[uuu]]]]][+HH:MM].
isoweekday()	Return the day of the week represented by the date.
now([tz])	Returns new datetime object representing current time local to tz.
replace()	Return datetime with new specified fields.
strftime()	format -> strftime() style string.

Continued on next page

Table 13 – continued from previous page

strptime()	string, format -> new datetime parsed from a string (like time.strptime()).
time()	Return time object with same time but with tzinfo=None.
timestamp()	Return POSIX timestamp as float.
timetuple()	Return time tuple, compatible with time.localtime().
timetz()	Return time object with same time and tzinfo.
today()	Current date or datetime: same as self.__class__.fromtimestamp(time.time()).
toordinal()	Return proleptic Gregorian ordinal.
tzname()	Return self.tzinfo.tzname(self).
utcfromtimestamp()	Construct a naive UTC datetime from a POSIX timestamp.
utcnow()	Return a new datetime representing UTC day and time.
utcoffset()	Return self.tzinfo.utcoffset(self).
utctimetuple()	Return UTC time tuple, compatible with time.localtime().
weekday()	Return the day of the week represented by the date.

astimezone()
tz -> convert to local time in new timezone tz

combine()
date, time -> datetime with same date and time fields

ctime()
Return ctime() style string.

date()
Return date object with same year, month and day.

dst()
Return self.tzinfo.dst(self).

fromisoformat()
string -> datetime from datetime.isoformat() output

fromtimestamp()
timestamp[, tz] -> tz's local time from POSIX timestamp.

isoformat()
[sep] -> string in ISO 8601 format, YYYY-MM-DDT[HH[:MM[:SS[.mmm[uuu]]]]][+HH:MM]. sep is used to separate the year from the time, and defaults to 'T'. timespec specifies what components of the time to include (allowed values are 'auto', 'hours', 'minutes', 'seconds', 'milliseconds', and 'microseconds').

now(*tz=None*)
Returns new datetime object representing current time local to tz.

tz Timezone object.

If no tz is specified, uses local timezone.

replace()
Return datetime with new specified fields.

strptime()
string, format -> new datetime parsed from a string (like time.strptime()).

time()
Return time object with same time but with tzinfo=None.

timestamp()
Return POSIX timestamp as float.

timetuple()
Return time tuple, compatible with time.localtime().

timetz()
Return time object with same time and tzinfo.

tzname()
Return self.tzinfo.tzname(self).

utcfromtimestamp()
Construct a naive UTC datetime from a POSIX timestamp.

utcnow()
Return a new datetime representing UTC day and time.

utcoffset()
Return self.tzinfo.utcoffset(self).

utctimetuple()
Return UTC time tuple, compatible with time.localtime().

mlrun.utils.**format_exception**(*etype*, *value*, *tb*, *limit=None*, *chain=True*)
Format a stack trace and the exception information.

The arguments have the same meaning as the corresponding arguments to print_exception(). The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does print_exception().

mlrun.utils.**get_in**(*obj*, *keys*, *default=None*)

```
>>> get_in({'a': {'b': 1}}, 'a.b')
1
```

mlrun.utils.**import_module**(*name*, *package=None*)
Import a module.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

mlrun.utils.**retry_until_successful**(*interval: int*, *timeout: int*, *logger*, *verbose: bool*, *_function*, **args*, ***kwargs*)
Runs function with given *args and **kwargs. Tries to run it until success or timeout reached (timeout is optional) :param interval: int/float that will be used as interval :param timeout: pass None if timeout is not wanted, number of seconds if it is :param logger: a logger so we can log the failures :param verbose: whether to log the failure on each retry :param _function: function to run :param args: functions args :param kwargs: functions kwargs :return: function result

mlrun.utils.**tabulate**(*tabular_data*, *headers=()*, *tablefmt='simple'*, *floatfmt='g'*, *numalign='decimal'*, *stralign='left'*, *missingval=''*, *showindex='default'*, *disable_numparse=False*, *colalign=None*)
Format a fixed width table for pretty printing.

```
>>> print(tabulate([[1, 2.34], [-56, "8.999"], ["2", "10001"]]))
---  ---------
```

(continues on next page)

(continued from previous page)

```
  1      2.34
-56      8.999
  2  10001
---  ---------
```

The first required argument (*tabular_data*) can be a list-of-lists (or another iterable of iterables), a list of named tuples, a dictionary of iterables, an iterable of dictionaries, a two-dimensional NumPy array, NumPy record array, or a Pandas' dataframe.

class mlrun.utils.**timezone**

Fixed offset from UTC implementation of tzinfo.

Methods

dst()	Return None.
fromutc()	datetime in UTC -> datetime in local time.
tzname()	If name is specified when timezone is created, returns the name.
utcoffset()	Return fixed offset.

dst()

Return None.

fromutc()

datetime in UTC -> datetime in local time.

tzname()

If name is specified when timezone is created, returns the name. Otherwise returns offset as 'UTC(+|-)HH:MM'.

utcoffset()

Return fixed offset.

See also the index of all functions and classes.

PYTHON MODULE INDEX

m

INDEX

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W