1-creating a workspace:

```from azureml.core import Workspace

ws = Workspace.create(name='myworkspace',

subscription\_id='<azure-subscription-id>',

resource\_group='myresourcegroup',

create\_resource\_group=True,

location='eastus2'

)```

Set create\_resource\_group to False if you have a previously existing Azure resource group that you want to use for the workspace.

2- saving workspace configuration as json:  
Use the same workspace in multiple environments by first writing it to a configuration JSON file. This saves your subscription, resource, and workspace name data.

```ws.write\_config(path="./file-path", file\_name="ws\_config.json")```

3.1-Load your workspace by reading the configuration file:

```from azureml.core import Workspace

ws\_other\_environment = Workspace.from\_config(path="./file-path/ws\_config.json")```

3.2 alternative method to load workspace using static method:

```from azureml.core import Workspace

ws = Workspace.get(name="myworkspace", subscription\_id='<azure-subscription-id>', resource\_group='myresourcegroup')```

4- Experiment:

The Experiment class is another foundational cloud resource that represents a collection of trials (individual model runs). The following code fetches an Experiment object from within Workspace by name, or it creates a new Experiment object if the name doesn't exist.

``` from azureml.core.experiment import Experiment

experiment = Experiment(workspace=ws, name='test-experiment')```

Run the following code to get a list of all Experiment objects contained in Workspace:

```list\_experiments = Experiment.list(ws)```

Use the get\_runs function to retrieve a list of Run objects (trials) from Experiment. The following code retrieves the runs and prints each run ID.

```list\_runs = experiment.get\_runs()

for run in list\_runs:

print(run.id)```

**Run**

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.run.run?view=azure-ml-py&preserve-view=true): azureml.core.run.Run

A run represents a single trial of an experiment. Run is the object that you use to monitor the asynchronous execution of a trial, store the output of the trial, analyze results, and access generated artifacts. You use Run inside your experimentation code to log metrics and artifacts to the Run History service. Functionality includes:

* Storing and retrieving metrics and data.
* Using tags and the child hierarchy for easy lookup of past runs.
* Registering stored model files for deployment.
* Storing, modifying, and retrieving properties of a run.

Create a Run object by submitting an Experiment object with a [run configuration](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.run(class)?view=azure-ml-py&preserve-view=true#remarks) object. Use the tags parameter to attach custom categories and labels to your runs. You can easily find and retrieve them later from Experiment.

```tags = {"prod": "phase-1-model-tests"}

run = experiment.submit(config=your\_config\_object, tags=tags)```

Use the static list function to get a list of all Run objects from Experiment. Specify the tags parameter to filter by your previously created tag.

```from azureml.core.run import Run

filtered\_list\_runs = Run.list(experiment, tags=tags)```

Use the get\_details function to retrieve the detailed output for the run

```run\_details = run.get\_details()```

Output for this function is a dictionary that includes:

* Run ID
* Status
* Start and end time
* Compute target (local versus cloud)
* Dependencies and versions used in the run
* Training-specific data (differs depending on model type)

## Model

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.model.model?view=azure-ml-py&preserve-view=true): azureml.core.model.Model

The Model class is used for working with cloud representations of machine learning models. Methods help you transfer models between local development environments and the Workspace object in the cloud.

You can use model registration to store and version your models in the Azure cloud, in your workspace. Registered models are identified by name and version. Each time you register a model with the same name as an existing one, the registry increments the version. Azure Machine Learning supports any model that can be loaded through Python 3, not just Azure Machine Learning models.

The following example shows how to build a simple local classification model with scikit-learn, register the model in Workspace, and download the model from the cloud.

Create a simple classifier, clf, to predict customer churn based on their age. Then dump the model to a .pkl file in the same directory.

```from sklearn import svm

import joblib

import numpy as np

# customer ages

X\_train = np.array([50, 17, 35, 23, 28, 40, 31, 29, 19, 62])

X\_train = X\_train.reshape(-1, 1)

# churn y/n

y\_train = ["yes", "no", "no", "no", "yes", "yes", "yes", "no", "no", "yes"]

clf = svm.SVC(gamma=0.001, C=100.)

clf.fit(X\_train, y\_train)

joblib.dump(value=clf, filename="churn-model.pkl")```

Use the register function to register the model in your workspace. Specify the local model path and the model name. Registering the same name more than once will create a new version.

```from azureml.core.model import Model

model = Model.register(workspace=ws, model\_path="churn-model.pkl", model\_name="churn-model-test")```

Now that the model is registered in your workspace, it's easy to manage, download, and organize your models. To retrieve a model (for example, in another environment) object from Workspace, use the class constructor and specify the model name and any optional parameters. Then, use the download function to download the model, including the cloud folder structure.

```from azureml.core.model import Model

import os

model = Model(workspace=ws, name="churn-model-test")

model.download(target\_dir=os.getcwd())```

Use the delete function to remove the model from Workspace.

```model.delete()```

After you have a registered model, deploying it as a web service is a straightforward process. First you [create and register an image](https://learn.microsoft.com/en-us/azure/machine-learning/how-to-deploy-and-where#configureimage). <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-deploy-online-endpoints?view=azureml-api-2&tabs=azure-cli#configureimage> This step configures the Python environment and its dependencies, along with a script to define the web service request and response formats. After you create an image, you [build a deploy configuration](https://learn.microsoft.com/en-us/azure/machine-learning/how-to-deploy-and-where#deploy)  <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-deploy-online-endpoints?view=azureml-api-2&tabs=azure-cli#deploy> that sets the CPU cores and memory parameters for the compute target. You then attach your image.

## ComputeTarget, RunConfiguration, and ScriptRunConfig

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.compute.computetarget?view=azure-ml-py&preserve-view=true): azureml.core.compute.ComputeTarget  
[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.runconfig.runconfiguration?view=azure-ml-py&preserve-view=true): azureml.core.runconfig.RunConfiguration  
[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.script_run_config.scriptrunconfig?view=azure-ml-py&preserve-view=true): azureml.core.script\_run\_config.ScriptRunConfig

The ComputeTarget class is the abstract parent class for creating and managing compute targets. A compute target represents a variety of resources where you can train your machine learning models. A compute target can be either a local machine or a cloud resource, such as Azure Machine Learning Compute, Azure HDInsight, or a remote virtual machine.

Use compute targets to take advantage of powerful virtual machines for model training, and set up either persistent compute targets or temporary runtime-invoked targets. For a comprehensive guide on setting up and managing compute targets, see the [how-to](https://learn.microsoft.com/en-us/azure/machine-learning/how-to-set-up-training-targets). <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-set-up-training-targets?view=azureml-api-1>

The following code shows a simple example of setting up an AmlCompute (child class of ComputeTarget) target. This target creates a runtime remote compute resource in your Workspace object. The resource scales automatically when a job is submitted. It's deleted automatically when the run finishes.

Reuse the simple scikit-learn churn model and build it into its own file, train.py, in the current directory. At the end of the file, create a new directory called outputs. This step creates a directory in the cloud (your workspace) to store your trained model that joblib.dump() serialized.

```# train.py

from sklearn import svm

import numpy as np

import joblib

import os

# customer ages

X\_train = np.array([50, 17, 35, 23, 28, 40, 31, 29, 19, 62])

X\_train = X\_train.reshape(-1, 1)

# churn y/n

y\_train = ["yes", "no", "no", "no", "yes", "yes", "yes", "no", "no", "yes"]

clf = svm.SVC(gamma=0.001, C=100.)

clf.fit(X\_train, y\_train)

os.makedirs("outputs", exist\_ok=True)

joblib.dump(value=clf, filename="outputs/churn-model.pkl")```

Next you create the compute target by instantiating a RunConfiguration object and setting the type and size. This example uses the smallest resource size (1 CPU core, 3.5 GB of memory). The list\_vms variable contains a list of supported virtual machines and their sizes.

```from azureml.core.runconfig import RunConfiguration

from azureml.core.compute import AmlCompute

list\_vms = AmlCompute.supported\_vmsizes(workspace=ws)

compute\_config = RunConfiguration()

compute\_config.target = "amlcompute"

compute\_config.amlcompute.vm\_size = "STANDARD\_D1\_V2"```

Create dependencies for the remote compute resource's Python environment by using the CondaDependencies class. The train.py file is using scikit-learn and numpy, which need to be installed in the environment. You can also specify versions of dependencies. Use the dependencies object to set the environment in compute\_config.

```from azureml.core.conda\_dependencies import CondaDependencies

dependencies = CondaDependencies()

dependencies.add\_pip\_package("scikit-learn")

dependencies.add\_pip\_package("numpy==1.15.4")

compute\_config.environment.python.conda\_dependencies = dependencies```

Now you're ready to submit the experiment. Use the ScriptRunConfig class to attach the compute target configuration, and to specify the path/file to the training script train.py. Submit the experiment by specifying the config parameter of the submit() function. Call wait\_for\_completion on the resulting run to see asynchronous run output as the environment is initialized and the model is trained.

```from azureml.core.experiment import Experiment

from azureml.core import ScriptRunConfig

script\_run\_config = ScriptRunConfig(source\_directory=os.getcwd(), script="train.py", run\_config=compute\_config)

experiment = Experiment(workspace=ws, name="compute\_target\_test")

run = experiment.submit(config=script\_run\_config)

run.wait\_for\_completion(show\_output=True)```

After the run finishes, the trained model file churn-model.pkl is available in your workspace.

**Environment**

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.environment?view=azure-ml-py&preserve-view=true): azureml.core.environment

Azure Machine Learning environments specify the Python packages, environment variables, and software settings around your training and scoring scripts. In addition to Python, you can also configure PySpark, Docker and R for environments. Internally, environments result in Docker images that are used to run the training and scoring processes on the compute target. The environments are managed and versioned entities within your Machine Learning workspace that enable reproducible, auditable, and portable machine learning workflows across a variety of compute targets and compute types.

You can use an Environment object to:

* Develop your training script.
* Reuse the same environment on Azure Machine Learning Compute for model training at scale.
* Deploy your model with that same environment without being tied to a specific compute type.

The following code imports the Environment class from the SDK and to instantiates an environment object.

```from azureml.core.environment import Environment

Environment(name="myenv")```

Add packages to an environment by using Conda, pip, or private wheel files. Specify each package dependency by using the [CondaDependency](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.conda_dependencies.condadependencies?view=azure-ml-py&preserve-view=true) class to add it to the environment's PythonSection.

The following example adds to the environment. It adds version 1.17.0 of numpy. It also adds the pillow package to the environment, myenv. The example uses the [add\_conda\_package()](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.conda_dependencies.condadependencies?view=azure-ml-py&preserve-view=true" \l "add-conda-package-conda-package-) method and the [add\_pip\_package()](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.conda_dependencies.condadependencies?view=azure-ml-py&preserve-view=true" \l "add-pip-package-pip-package-) method, respectively.

```from azureml.core.environment import Environment

from azureml.core.conda\_dependencies import CondaDependencies

myenv = Environment(name="myenv")

conda\_dep = CondaDependencies()

# Installs numpy version 1.17.0 conda package

conda\_dep.add\_conda\_package("numpy==1.17.0")

# Installs pillow package

conda\_dep.add\_pip\_package("pillow")

# Adds dependencies to PythonSection of myenv

myenv.python.conda\_dependencies=conda\_dep```

To submit a training run, you need to combine your environment, [compute target](https://learn.microsoft.com/en-us/azure/machine-learning/concept-compute-target), and your training Python script into a run configuration. This configuration is a wrapper object that's used for submitting runs.

When you submit a training run, the building of a new environment can take several minutes. The duration depends on the size of the required dependencies. The environments are cached by the service. So as long as the environment definition remains unchanged, you incur the full setup time only once.

The following example shows where you would use [ScriptRunConfig](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.script_run_config.scriptrunconfig?view=azure-ml-py&preserve-view=true) as your wrapper object.

```from azureml.core import ScriptRunConfig, Experiment

from azureml.core.environment import Environment

exp = Experiment(name="myexp", workspace = ws)

# Instantiate environment

myenv = Environment(name="myenv")

# Add training script to run config

runconfig = ScriptRunConfig(source\_directory=".", script="train.py")

# Attach compute target to run config

runconfig.run\_config.target = "local"

# Attach environment to run config

runconfig.run\_config.environment = myenv

# Submit run

run = exp.submit(runconfig)```

If you don't specify an environment in your run configuration before you submit the run, then a default environment is created for you.

See the [Model deploy](https://learn.microsoft.com/en-us/python/api/overview/azure/ml/?view=azure-ml-py#model-deploy)  <https://learn.microsoft.com/en-us/python/api/overview/azure/ml/?view=azure-ml-py#model-deploy> section to use environments to deploy a web service.

**Pipeline, PythonScriptStep**

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-pipeline-core/azureml.pipeline.core.pipeline.pipeline?view=azure-ml-py&preserve-view=true): azureml.pipeline.core.pipeline.Pipeline  
[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-pipeline-steps/azureml.pipeline.steps.python_script_step.pythonscriptstep?view=azure-ml-py&preserve-view=true): azureml.pipeline.steps.python\_script\_step.PythonScriptStep

An Azure Machine Learning pipeline is an automated workflow of a complete machine learning task. Subtasks are encapsulated as a series of steps within the pipeline. An Azure Machine Learning pipeline can be as simple as one step that calls a Python script. Pipelines include functionality for:

* Data preparation including importing, validating and cleaning, munging and transformation, normalization, and staging
* Training configuration including parameterizing arguments, filepaths, and logging / reporting configurations
* Training and validating efficiently and repeatably, which might include specifying specific data subsets, different hardware compute resources, distributed processing, and progress monitoring
* Deployment, including versioning, scaling, provisioning, and access control
* Publishing a pipeline to a REST endpoint to rerun from any HTTP library

A PythonScriptStep is a basic, built-in step to run a Python Script on a compute target. It takes a script name and other optional parameters like arguments for the script, compute target, inputs and outputs. The following code is a simple example of a PythonScriptStep. For an example of a train.py script, see the [tutorial](https://learn.microsoft.com/en-us/azure/machine-learning/tutorial-pipeline-batch-scoring-classification#write-a-scoring-script)  <https://learn.microsoft.com/en-us/azure/machine-learning/tutorial-pipeline-python-sdk?view=azureml-api-2#write-a-scoring-script> sub-section.

```from azureml.pipeline.steps import PythonScriptStep

train\_step = PythonScriptStep(

script\_name="train.py",

arguments=["--input", blob\_input\_data, "--output", output\_data1],

inputs=[blob\_input\_data],

outputs=[output\_data1],

compute\_target=compute\_target,

source\_directory=project\_folder

)```

After at least one step has been created, steps can be linked together and published as a simple automated pipeline.

from azureml.pipeline.core import Pipeline

pipeline = Pipeline(workspace=ws, steps=[train\_step])

pipeline\_run = experiment.submit(pipeline)

For a comprehensive example of building a pipeline workflow, follow the [advanced tutorial](https://learn.microsoft.com/en-us/azure/machine-learning/tutorial-pipeline-batch-scoring-classification). <https://learn.microsoft.com/en-us/azure/machine-learning/tutorial-pipeline-python-sdk?view=azureml-api-2>

### Pattern for creating and using pipelines

An Azure Machine Learning pipeline is associated with an Azure Machine Learning workspace and a pipeline step is associated with a compute target available within that workspace. For more information, see this [article about workspaces](https://learn.microsoft.com/en-us/azure/machine-learning/how-to-manage-workspace) <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-manage-workspace?view=azureml-api-2&tabs=python>  or this explanation of [compute targets](https://learn.microsoft.com/en-us/azure/machine-learning/concept-compute-target).

A common pattern for pipeline steps is:

1. Specify workspace, compute, and storage
2. Configure your input and output data using
   1. [Dataset](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.dataset.dataset?view=azure-ml-py&preserve-view=true) which makes available an existing Azure datastore
   2. [PipelineDataset](https://learn.microsoft.com/en-us/python/api/azureml-pipeline-core/azureml.pipeline.core.pipelinedataset?view=azure-ml-py&preserve-view=true) which encapsulates typed tabular data
   3. [PipelineData](https://learn.microsoft.com/en-us/python/api/azureml-pipeline-core/azureml.pipeline.core.pipelinedata?view=azure-ml-py&preserve-view=true) which is used for intermediate file or directory data written by one step and intended to be consumed by another
3. Define one or more pipeline steps
4. Instantiate a pipeline using your workspace and steps
5. Create an experiment to which you submit the pipeline
6. Monitor the experiment results

[This notebook](https://github.com/Azure/MachineLearningNotebooks/blob/master/how-to-use-azureml/machine-learning-pipelines/intro-to-pipelines/aml-pipelines-getting-started.ipynb) <https://github.com/Azure/MachineLearningNotebooks/blob/master/how-to-use-azureml/machine-learning-pipelines/intro-to-pipelines/aml-pipelines-getting-started.ipynb>  is a good example of this pattern. job

For more information about Azure Machine Learning Pipelines, and in particular how they are different from other types of pipelines, see this [article](https://learn.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines). <https://learn.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines?view=azureml-api-2>

## AutoMLConfig

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-train-automl-client/azureml.train.automl.automlconfig.automlconfig?view=azure-ml-py&preserve-view=true): azureml.train.automl.automlconfig.AutoMLConfig

Use the AutoMLConfig class to configure parameters for automated machine learning training. Automated machine learning iterates over many combinations of machine learning algorithms and hyperparameter settings. It then finds the best-fit model based on your chosen accuracy metric. Configuration allows for specifying:

* Task type (classification, regression, forecasting)
* Number of algorithm iterations and maximum time per iteration
* Accuracy metric to optimize
* Algorithms to blocklist/allowlist
* Number of cross-validations
* Compute targets
* Training data

**Note**

Use the automl extra in your installation to use automated machine learning.

For detailed guides and examples of setting up automated machine learning experiments, see the [tutorial](https://learn.microsoft.com/en-us/azure/machine-learning/tutorial-auto-train-models) <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-auto-train-models-v1?view=azureml-api-1>  and [how-to](https://learn.microsoft.com/en-us/azure/machine-learning/how-to-configure-auto-train). <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-configure-auto-train?view=azureml-api-2>

The following code illustrates building an automated machine learning configuration object for a classification model, and using it when you're submitting an experiment.

```from azureml.train.automl import AutoMLConfig

automl\_config = AutoMLConfig(task="classification",

X=your\_training\_features,

y=your\_training\_labels,

iterations=30,

iteration\_timeout\_minutes=5,

primary\_metric="AUC\_weighted",

n\_cross\_validations=5

)```

Use the automl\_config object to submit an experiment.

```from azureml.core.experiment import Experiment

experiment = Experiment(ws, "automl\_test\_experiment")

run = experiment.submit(config=automl\_config, show\_output=True)```

After you submit the experiment, output shows the training accuracy for each iteration as it finishes. After the run is finished, an AutoMLRun object (which extends the Run class) is returned. Get the best-fit model by using the get\_output() function to return a Model object.

```best\_model = run.get\_output()

y\_predict = best\_model.predict(X\_test)```

## Model deploy

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.model.inferenceconfig?view=azure-ml-py&preserve-view=true): azureml.core.model.InferenceConfig  
[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.webservice.webservice.webservice?view=azure-ml-py&preserve-view=true): azureml.core.webservice.webservice.Webservice

The InferenceConfig class is for configuration settings that describe the environment needed to host the model and web service.

Webservice is the abstract parent class for creating and deploying web services for your models. For a detailed guide on preparing for model deployment and deploying web services, see this [how-to](https://learn.microsoft.com/en-us/azure/machine-learning/how-to-deploy-and-where). <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-deploy-online-endpoints?view=azureml-api-2>

You can use [environments](https://learn.microsoft.com/en-us/python/api/overview/azure/ml/?view=azure-ml-py#environment) when you deploy your model as a web service. Environments enable a reproducible, connected workflow where you can deploy your model using the same libraries in both your training compute and your inference compute. Internally, environments are implemented as Docker images. You can use either images provided by Microsoft, or use your own custom Docker images. If you were previously using the ContainerImage class for your deployment, see the [DockerSection](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.environment.dockersection?view=azure-ml-py&preserve-view=true) class for accomplishing a similar workflow with environments.

To deploy a web service, combine the environment, inference compute, scoring script, and registered model in your deployment object, [deploy()](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.model.model?view=azure-ml-py&preserve-view=true#deploy-workspace--name--models--inference-config-none--deployment-config-none--deployment-target-none--overwrite-false-).

The following example, assumes you already completed a training run using environment, myenv, and want to deploy that model to Azure Container Instances.

```from azureml.core.model import InferenceConfig, Model

from azureml.core.webservice import AciWebservice, Webservice

# Register the model to deploy

model = run.register\_model(model\_name = "mymodel", model\_path = "outputs/model.pkl")

# Combine scoring script & environment in Inference configuration

inference\_config = InferenceConfig(entry\_script="score.py",

environment=myenv)

# Set deployment configuration

deployment\_config = AciWebservice.deploy\_configuration(cpu\_cores = 1,

memory\_gb = 1)

# Define the model, inference, & deployment configuration and web service name and location to deploy

service = Model.deploy(workspace = ws,

name = "my\_web\_service",

models = [model],

inference\_config = inference\_config,

deployment\_config = deployment\_config)```

This example creates an Azure Container Instances web service, which is best for small-scale testing and quick deployments. To deploy your model as a production-scale web service, use Azure Kubernetes Service (AKS). For more information, see [AksCompute class](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.compute.akscompute?view=azure-ml-py&preserve-view=true).

**Dataset**

[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.dataset.dataset?view=azure-ml-py&preserve-view=true): azureml.core.dataset.Dataset  
[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.data.file_dataset.filedataset?view=azure-ml-py&preserve-view=true): azureml.data.file\_dataset.FileDataset  
[Namespace](https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.data.tabular_dataset.tabulardataset?view=azure-ml-py&preserve-view=true): azureml.data.tabular\_dataset.TabularDataset

The Dataset class is a foundational resource for exploring and managing data within Azure Machine Learning. You can explore your data with summary statistics, and save the Dataset to your AML workspace to get versioning and reproducibility capabilities. Datasets are easily consumed by models during training. For detailed usage examples, see the [how-to guide](https://learn.microsoft.com/en-us/azure/machine-learning/how-to-create-register-datasets). <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-create-data-assets?view=azureml-api-2&tabs=cli>

* TabularDataset represents data in a tabular format created by parsing a file or list of files.
* FileDataset references single or multiple files in datastores or from public URLs.

The following example shows how to create a TabularDataset pointing to a single path in a datastore.

```from azureml.core import Dataset

dataset = Dataset.Tabular.from\_delimited\_files(path = [(datastore, 'train-dataset/tabular/iris.csv')])

dataset.take(3).to\_pandas\_dataframe()```

The following example shows how to create a FileDataset referencing multiple file URLs.

```from azureml.core.dataset import Dataset

url\_paths = [

'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',

'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',

'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',

'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'

]

dataset = Dataset.File.from\_files(path=url\_paths)```