**Metaflow Artifacts and Parameters**

In **Metaflow**, **artifacts** and **parameters** are two critical features that enable the sharing of data between steps and the customization of flow execution. Understanding how to handle both is important when building scalable data pipelines or machine learning workflows.

**1. Artifacts:**

**Artifacts** are variables that you want to persist and make available across different steps of a flow. Any object (e.g., data, models, or configurations) assigned to a class attribute during a step can be stored as an artifact, which will then be accessible in subsequent steps.

**Defining Artifacts:**

Artifacts are defined by assigning values to self within a step. Once an artifact is defined, it can be accessed in any downstream step.

**Example of Using Artifacts:**

from metaflow import FlowSpec, step

class ArtifactFlow(FlowSpec):

@step

def start(self):

# Define an artifact (self.x)

self.x = 10

print(f"Start step, self.x = {self.x}")

self.next(self.middle)

@step

def middle(self):

# Access the artifact from the previous step

self.y = self.x + 20

print(f"Middle step, self.y = {self.y}")

self.next(self.end)

@step

def end(self):

# Access all previous artifacts

print(f"End step, self.x = {self.x}, self.y = {self.y}")

print("Flow completed")

In this flow:

* self.x is stored as an artifact in the start step.
* In the middle step, the artifact self.x is accessed and used to create another artifact, self.y.
* Both artifacts are accessible in the end step.

**Key Features:**

* Artifacts are automatically persisted between steps.
* They can be complex data structures like lists, dictionaries, DataFrames, or machine learning models.

**2. Parameters:**

**Parameters** allow users to specify values when launching a flow. This makes the flow more dynamic, as different runs can use different input values without modifying the code. Parameters are especially useful for configuration settings, input data paths, or hyperparameters in machine learning models.

**Defining Parameters:**

Parameters are declared using the @parameter decorator. They can be set via the command line when running the flow or given default values.

**Example of Using Parameters:**

from metaflow import FlowSpec, step, Parameter

class ParameterFlow(FlowSpec):

# Define a parameter with a default value

multiplier = Parameter('multiplier', default=2, help="Multiplier value")

@step

def start(self):

# Access the parameter value

self.x = 10

print(f"Start step, self.x = {self.x}, multiplier = {self.multiplier}")

self.next(self.multiply)

@step

def multiply(self):

# Use the parameter in computations

self.result = self.x \* self.multiplier

print(f"Multiply step, result = {self.result}")

self.next(self.end)

@step

def end(self):

print(f"End step, final result = {self.result}")

print("Flow completed")

In this flow:

* The multiplier parameter is declared with a default value of 2.
* The value of multiplier can be passed via the command line when executing the flow, or the default value will be used.
* The parameter is accessed in the multiply step to calculate a result.

**Running with Custom Parameters:**

You can provide custom values for parameters when running the flow from the command line:

python parameter\_flow.py run --multiplier 5

This will override the default value and use 5 as the multiplier.

**Key Features:**

* Parameters can be integers, floats, strings, lists, or more complex types.
* Parameters allow flexibility when rerunning flows with different configurations.

**3. Combining Artifacts and Parameters:**

You can combine both **artifacts** and **parameters** in a single flow to create more dynamic and flexible workflows. Parameters can be used to control the flow behavior, while artifacts are used to persist and share data between steps.

**Example:**

from metaflow import FlowSpec, step, Parameter

class DynamicFlow(FlowSpec):

# Define a parameter to control the computation

multiplier = Parameter('multiplier', default=2)

@step

def start(self):

# Define an artifact

self.x = 10

print(f"Start step, self.x = {self.x}, multiplier = {self.multiplier}")

self.next(self.multiply)

@step

def multiply(self):

# Access the parameter and use it in calculations

self.result = self.x \* self.multiplier

print(f"Multiply step, result = {self.result}")

self.next(self.end)

@step

def end(self):

print(f"End step, final result = {self.result}")

print("Flow completed")

This flow:

* Uses a parameter multiplier to control how the artifact self.x is processed in the multiply step.
* The result is persisted as an artifact and printed in the end step.

**Summary of Best Practices:**

* **Artifacts**:
  + Use artifacts (self.<variable>) to persist data between steps in a flow.
  + Artifacts should be used for sharing data that will be reused by subsequent steps.
* **Parameters**:
  + Use parameters to define inputs or settings that may change across different runs (e.g., hyperparameters for a model).
  + Parameters add flexibility to the flow by allowing customization at runtime without modifying code.

**Use Case Example:**

For a machine learning workflow:

* You might define a parameter for the number of training epochs or the learning rate.
* Artifacts can store the trained model, training data, and metrics, making them available in subsequent steps for evaluation, prediction, or deployment.

**Summary Table:**

| **Concept** | **Purpose** | **Example** |
| --- | --- | --- |
| **Artifact** | Persist and share data between steps | self.x = 10 |
| **Parameter** | Customize flow behavior at runtime | multiplier = Parameter('multiplier', default=2) |
| **Usage** | Artifacts store data, parameters customize the flow | Flow customization at runtime based on inputs and persisted data |

By effectively using both artifacts and parameters, you can build flexible, dynamic, and reusable workflows in Metaflow.