Loading Palantir Foundry Data into Python

DATA ENGINEERING ARCHITECTURE HANDOUT

Palantir Foundry functions as an enterprise data platform. Programmatic access to its datasets depends entirely on context: whether computation executes **externally** (local Anaconda/IDE environments leveraging RESTful APIs) or **internally** (natively containerized within Foundry's Spark-backed infrastructure).

ARCHITECTURAL LOADING METHODS

Method 1: External Client Integration (Local Anaconda Environments)

To ingest enterprise data down into an off-platform script or localized notebook environment, utilize the verified API-backed **Palantir Foundry Python SDK**.

Step 1: Install Dependencies via Terminal

```
# Activate your data science project environment context
conda activate genai_env

# Package synchronization using pip to retrieve the core foundry API client
pip install palantir-foundry-client
```

Step 2: Generate Authentication Access

Navigate to your user account profile within Foundry → Account Settings → Personal Access Tokens → Generate Token. Copy and protect this explicit Bearer signature sequence.

Step 3: Programmatic Extraction Script Pattern

```
from foundry.v2 import FoundryClient

# 1. Authorize the REST client interface
client = FoundryClient(
    hostname="your-organization.palantirfoundry.com",
    auth="YOUR_PERSONAL_ACCESS_TOKEN"
)

# 2. Specify the target Resource Identifier (RID) coordinates
dataset_rid = "ri.foundry.main.dataset.xxxx-xxxx-xxxx-xxxx"

# 3. Stream data streams out of file nodes directly into Pandas memory structures
dataset = client.datasets.Dataset.read(dataset_rid)
df = dataset.to_pandas()

print(f"Loaded dataset containing {len(df)} rows.")
print(df.head())
```

Method 2: Internal Native Workflows (Foundry Code Workbooks)

When engineering within a **Code Workbook** on the cluster platform, datasets don't require API connections. Instead, datasets are added visually to a graphical data lineage canvas via drag-and-drop mechanics.

Foundry automatically streams dependencies straight into code cells as a configured object state:

Alternative A: Distributed Big-Data Configurations (Spark DataFrame)

```
def transform_logic(input_dataset):
    # Data arrives pre-loaded as an implicit PySpark DataFrame reference
    df = input_dataset

    # Perform performant columnar analytical filtering
    processed_df = df.filter(df["status"] == "active")

    return processed_df
```

Alternative B: Standard Data Matrices (Pandas DataFrame)

```
def transform_logic(input_dataset):
    # Delivered as a local pandas matrix for lightweight arrays
    print(f"Inspecting active data headers: {input_dataset.columns}")
    return input_dataset
```

Method 3: Direct Relational Drivers (ODBC/JDBC Layer)

For applications heavily reliant on relational query syntax frameworks (such as `pyodbc`, `SQLAlchemy`, or `Ibis`), connection relies on downloading Palantir's proprietary database communication binaries.

```
import pandas as pd
import pyodbc

# Connect using your system-configured Data Source Name (DSN) setup
conn = pyodbc.connect("DSN=Foundry_SQL_Connection;UID=user;PWD=your_token")

sql_query = "SELECT * FROM `/Company/Folder/My_Dataset` WHERE year = 2026"
df = pd.read_sql(sql_query, conn)
```

⚠️ ENTERPRISE PRODUCTION BEST PRACTICES

- **Prune Early:** Enterprise datasets often scale to hundreds of millions of rows. Never load an unpartitioned dataset blindly into local RAM. Apply server-side operations (SQL slicing or API column selecting) *prior* to serialization into local space.
- **Optimal Serializers:** Where supported by the client SDK layer, utilize binary parquet columnar loaders or Apache Arrow streaming instead of heavy plain-text CSV formats to safeguard network bandwidth.