

Datastar Python Library



Table of Contents

- [Description](#)
- [Use cases](#)
- [Installation](#)
 - [Authentication](#)
 - [Running on the Optilogic platform](#)
 - [Working on the desktop](#)
 - [Working with the library](#)
- [Projects](#)
- [Macros](#)
- [Tasks](#)
 - [Joining tasks together](#)
 - [Task types](#)
 - [Connections](#)
 - [Sandbox access with pandas](#)
- [Example end-to-end flow](#)

Description

This library provides helper functions for working with projects and data in Optilogic Datastar.

The following documentation explains how to authenticate, connect to Datastar projects, and use high-level helpers for macros, tasks, and connections to build your own workflows with minimal boilerplate.

Use cases

The purpose of the library is to simplify creating and managing Datastar projects, defining macros composed of tasks, and moving data between sources and destinations. Typical objectives include:

- Connect to your own data sources.
- Build macros programmatically with import/export, SQL, and Python tasks.
- Chain tasks, then run and monitor macro executions.
- Read from and write to the project sandbox using pandas.

Some simple code examples show the python interface:

Connect to an existing project by name:

```
from datastar import *  
  
project = Project.connect_to("My Existing Project")
```

List your projects:

```
from datastar import *  
  
project_names = Project.get_projects()  
print(project_names)
```

Create a new project:

```
from datastar import *  
  
project = Project.create("My New Project", description="Example project")
```

Installation

Install via pip:

```
pip install ol-datastar
```

Import the high-level interfaces:

```
from datastar import *
```

Authentication

Datastar requires an App Key for API access.

Running on the Optilogic platform

When your code runs on the Optilogic platform, your credentials are auto-detected. Simply import and use the library:

```
from datastar import *  
  
project = Project.connect_to("My Existing Project")
```

Working on the desktop

When running locally, place your App Key in a file named **app.key** in the same directory as your script. The library reads this file automatically:

1. Create an App Key in the UI: <https://optilogic.app/#/user-account?tab=appkey>
2. Copy the key and paste it into a file called **app.key** next to your script.
3. Run your script as normal.

```
my_script.py  
app.key
```

Warning: Keep App Keys secure. Do not commit **app.key** to source control or share it publicly.

Working with the library

The library aims to be concise and discoverable. Most editors show tooltips for arguments and return types:

```
(method) def read_table(  
    table_name: str,  
    id_col: bool = False  
) -> DataFrame
```

Read a single model table and return as a DataFrame

Args:

- table_name: Table name to be read (supporting custom tables)
- id_col: Indicates whether the table id column should be returned

Returns:

- Single dataframe holding table contents

Projects

Projects are the top-level container for your macros and connections.

List all accessible projects:

```
from datastar import *  
  
names = Project.get_projects()  
print(names)
```

Connect to an existing project by name:

```
from datastar import *  
  
project = Project.connect_to("My Existing Project")
```

Create a new project (optionally with a description):

```
from datastar import *  
  
project = Project.create("My New Project", description="Example project")
```

Rename and update a project:

```
from datastar import *

project = Project.connect_to("My Existing Project")
project.name = "Updated name"
project.description = "Updated description"
project.save()
```

Delete a project:

```
from datastar import *

project = Project.connect_to("My Existing Project")
project.delete()
```

Macros

Macros are workflows composed of ordered tasks.

Add a new macro to a project:

```
from datastar import *

project = project.connect_to("My project")

macro = project.add_macro(name="load_and_process", description="ETL pipeline")
```

Get a list of macro names in a project:

```
from datastar import *

project = project.connect_to("My project")
macro_names = project.get_macros()
print(macro_names)
```

Get an existing macro in a project:

```
from datastar import *

project = project.connect_to("My project")
macro = project.get_macro("load_and_process")
```

Delete a macro, from a project:

```
from datastar import *

project = project.connect_to("My project")

project.delete_macro("old_macro")
```

Delete a macro (alternative, from macro):

```
from datastar import *

project = project.connect_to("My project")
macro = project.get_macro("load_and_process")
macro.delete()
```

Change macro name or description

```
from datastar import *

project = project.connect_to("My project")
macro = project.get_macro("load_and_process")
macro.name = "pipeline"
macro.description = "Daily ETL"
macro.save()
```


Copying tasks between macros

You can copy an existing task either within the same macro or into a different macro. Use

`macro.add_task(existing_task, ...)`:

```
from datastar import *

project = Project.connect_to("My project")
macro = project.get_macro("load_and_process")

# Suppose sql_task is an existing task in this macro
sql_task = macro.add_run_sql_task(
    name="cleanup",
    connection=project.get_sandbox(),
    query="DELETE FROM t WHERE id IS NULL",
)

# 1) Copy the task within the same macro
# The library creates a new task with name suffixed by "(copy)"
sql_task_copy_same = macro.add_task(sql_task)

# 2) Copy the task into a different macro in the same project
target_macro = project.add_macro("copy_target")
sql_task_copy_other = target_macro.add_task(sql_task)

# Optional: control joining behavior
# - auto_join=True chains after the last task added to the target macro
# - or specify previous_task to connect explicitly
start_task = target_macro.get_task("Start")
explicit_copy = target_macro.add_task(sql_task, auto_join=False,
previous_task=start_task)
```

Notes:

- The original task is never modified; a new task is created in the target macro.
- When copying within the same macro, the new task name is automatically suffixed with "(copy)" to avoid collisions.

Cloning macros

Clone an entire macro (all non-Start tasks and their dependencies) either within the same project or into a different project using `macro.clone(...)`:

```

from datastar import *

project = Project.connect_to("My project")
macro = project.get_macro("load_and_process")

# 1) Clone within the same project
# The new macro name defaults to "<name> (copy)"
macro_clone_same = macro.clone()

# 2) Clone into a new project
dest_project = Project.create("Clone Target Project")
macro_clone_other = macro.clone(project=dest_project)

# Optional parameters allow overriding name/description
macro_custom = macro.clone(name="pipeline_clone", description="Cloned with
custom name")

```

Notes:

- Each destination macro has its own Start task; Start is not duplicated.
- Task configurations are copied, and dependencies are recreated between the copied tasks.

Run a macro:

```

from datastar import *

# ... project and macro setup here
run_id = macro.run(parameters={})

```

Wait for a macro to complete:

```

from datastar import *

# ... project and macro setup here
run_id = macro.run(parameters={})
macro.wait_for_done(run_id, verbose=True)

```

Note that the verbose parameter controls whether updates are printed.

Tasks

Tasks within macros describe the actions that will be performed.

Initially a macro has only a start node, and tasks created by the user are generally connected to this in a chain or graph.

Joining tasks together

By default when the library adds a task it will be joined onto the last task added to the macro (or 'Start' if is the first Task to be added).

This behaviour can be overridden.

When creating a task, use `auto_join=False` and pass the `previous_task` parameter.

If no previous task is supplied, then the task will not be connected.

Tasks can be connected and disconnected from their predecessor using `task.add_dependency(previous_task_name)` and `task.remove_dependency(previous_task_name)`.

To get a list of task predecessors:

```
deps = sql_task.get_dependencies()
print(deps)
```

Task types

ImportTask — Use an ImportTask to load data into the project environment.

```
from datastar import *

macro = project.add_macro(name="load_and_process")
src = DelimitedConnection(path="/My Files/Datastar/Customers.csv",
delimiter=",")
dst = project.get_sandbox() # special in-project sandbox connection

import_task = macro.add_import_task(
    name="import_customers",
    source_connection=src,
    destination_connection=dst,
    destination_table="customers_raw",
    destination_table_type="new",          # "new" or "existing"
    destination_table_action="replace",    # "replace" or "append"
    condition="",                          # optional WHERE filter
    mappings=[("source_col", "dest_col")], # optional column mappings
)
```

RunSQLTask — Use a RunSQL task to execute SQL against a connection (e.g., sandbox).

```
sql_task = macro.add_run_sql_task(
    name="cleanup",
    connection=dst,
    query="DELETE FROM customers_raw WHERE city = 'London'",
)
```

RunPythonTask — Use a RunPythonTask to execute a Python file available to the platform.

```
py_task = macro.add_run_python_task(
    name="python_step",
    filename="my_script.py",
    directory_path="/My Files/Python coding",
)
```

ExportTask — Use an ExportTask to write data out to an external destination.

```

export_task = macro.add_export_task(
    name="export_results",
    source_connection=dst,
    source_table="customers_raw",
    file_name="Customers.csv",
)

```

Connections

Connections define sources/destinations for tasks and allow you to connect to different kinds of data sources.

Note: For **DelimitedConnection** and **ExcelConnection**, **path** refers to a path accessible to the platform (e.g., under "My Files").

DelimitedConnection - Use a DelimitedConnection for CSV files or other delimited file types.

```

from datastar import *

csv = DelimitedConnection(
    path="/My Files/Datastar/Customers.csv",
    delimiter=",",
)

```

ExcelConnection - Use an ExcelConnection to access .xls and .xlsx files.

```

from datastar import *

xl = ExcelConnection(
    path="/My Files/Datastar/Workbook.xlsx", # path to file on Optilogic
    workspace
    sheet_name="Sheet1",                    # optional
    workbook_password=None,                 # optional, can be supplied if
    password required.
)

```

FrogModelConnection - Use a FrogModelConnection to access data in a Cosmic Frog model.

```

from datastar import *

frog = FrogModelConnection(
    model_name="My Frog Model"              # or use storage_id="..."
)

```

Rename a connection:

```
csv.rename("My CSV Source")
```

Delete a connection:

```
csv.delete()
```

Sandbox access with pandas

The Sandbox connection exposes convenient helpers for direct table I/O using pandas. This is useful for quick inspection and ad-hoc data loads independent of macros.

```
import pandas as pd

sandbox = project.get_sandbox()

# Read a table
df = sandbox.read_table("customers_raw")

# Write a DataFrame
sandbox.write_table(df, "customers_raw")
```

Behind the scenes this uses SQLAlchemy with credentials provided by your App Key.

Example end-to-end flow

```
from datastar import *

project = Project.create("Datastar Demo")
macro = project.add_macro(name="pipeline")

src = DelimitedConnection(path="/My Files/Datastar/Customers.csv")
dst = project.get_sandbox()

macro.add_import_task(
    name="import_customers",
    source_connection=src,
    destination_connection=dst,
    destination_table="customers_raw",
)

macro.add_run_sql_task(
    name="dedupe",
    connection=dst,
    query="""
        CREATE TABLE IF NOT EXISTS customers AS
        SELECT DISTINCT * FROM customers_raw;
    """,
)

run_id = macro.run()
macro.wait_for_done(run_id, verbose=True)
```

Happy building! For additional help, contact support or see Optilogic resources at <https://www.optilogic.com/>.

API Summary

Project

Properties

- name — Display name of the project.
- description — Human-readable description for the project.

Methods

- get_projects() — List project names you can access.
- create(name, description) — Create a new project.
- connect_to(name) — Connect to an existing project by name.
- save() — Persist name/description changes.
- delete() — Remove the project.
- get_macros() — List macro names in the project.
- add_macro(name, description) — Create and add a macro.
- get_macro(name) — Get a macro by name.
- delete_macro(name) — Delete a macro by name.
- get_sandbox() — Get the project's Sandbox connection.

Macro

Properties

- project — The owning project.
- name — Display name of the macro.
- description — Human-readable description for the macro.

Methods

- save() — Persist name/description changes.
- delete() — Remove the macro from the project.
- get_task(name) — Get a task by name.
- get_tasks(type_filter) — List task names, optionally by type.
- delete_task(name) — Delete a task by name.
- add_task(task, auto_join, previous_task) — Add a pre-built task to the macro (copy tasks within/across macros; same-macro copies get a "(copy)" suffix).
- add_import_task(...) — Create and add an Import task.
- add_export_task(...) — Create and add an Export task.
- add_run_sql_task(...) — Create and add a Run SQL task.
- add_run_python_task(...) — Create and add a Run Python task.
- clone(project=None, name=None, description=None) — Clone the macro (tasks+dependencies) within the same or into another project.
- run(parameters) — Execute the macro and return run id.
- get_run_status(run_id) — Get the status of a run.

- `wait_for_done(run_id, verbose)` — Wait until a run completes.

Connection

Properties

- `name` — Display name of the connection.
- `description` — Human-readable description for the connection.

Methods

- `get_connections(connector_type)` — List connection names, optionally by type.
- `save()` — Persist connection changes.
- `delete()` — Remove the connection.

Task

Properties

- `macro` — The owning macro.
- `name` — Display name of the task.
- `description` — Human-readable description for the task.

Methods

- `save(auto_join, previous_task)` — Create or update the task.
- `delete()` — Remove the task from the macro.
- `add_dependency(previous_task_name)` — Connect this task after another.
- `remove_dependency(previous_task_name)` — Disconnect from a previous task.
- `get_dependencies()` — List names of predecessor tasks.