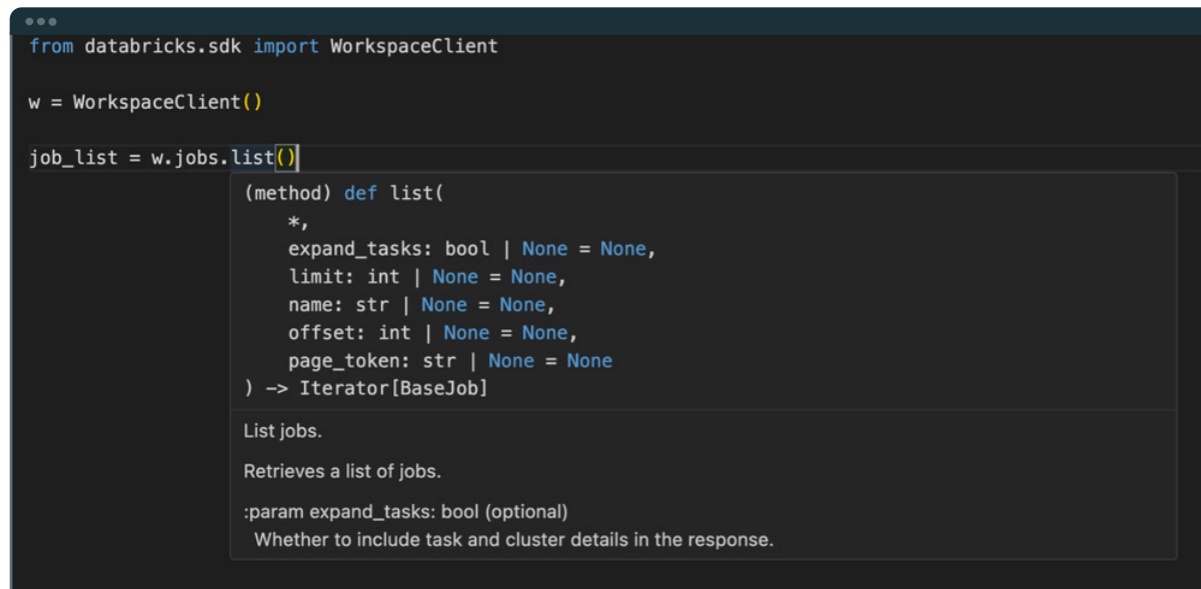


This is where IntelliSense really comes in handy. Instead of context switching between the IDE and the Documentation page, I can use autocomplete to provide a list of methods as well as examine the method description, the parameters and return types from within the IDE. I know the first step is getting a list of all the jobs in the workspace:



```

from databricks.sdk import WorkspaceClient

w = WorkspaceClient()

job_list = w.jobs.list()
  
```

(method) def list(
 *,
 expand_tasks: bool | None = None,
 limit: int | None = None,
 name: str | None = None,
 offset: int | None = None,
 page_token: str | None = None
) -> Iterator[BaseJob]

List jobs.
 Retrieves a list of jobs.

:param expand_tasks: bool (optional)
 Whether to include task and cluster details in the response.

As you can see, it returns an iterator over an object called BaseJob. Before we talk about what a BaseJob actually is, it'll be helpful to understand how data is used in the SDK. To interact with data you are sending to and receiving from the API, the Python SDK takes advantage of Python **data classes** and **enums**. The main advantage of this approach over passing around dictionaries is improved readability while also minimizing errors through enforced type checks and validations.

You can construct objects with data classes and interact with enums.

For example:

- *Creating an Employee via Employee DataClass and company departments, using enums for possible department values*

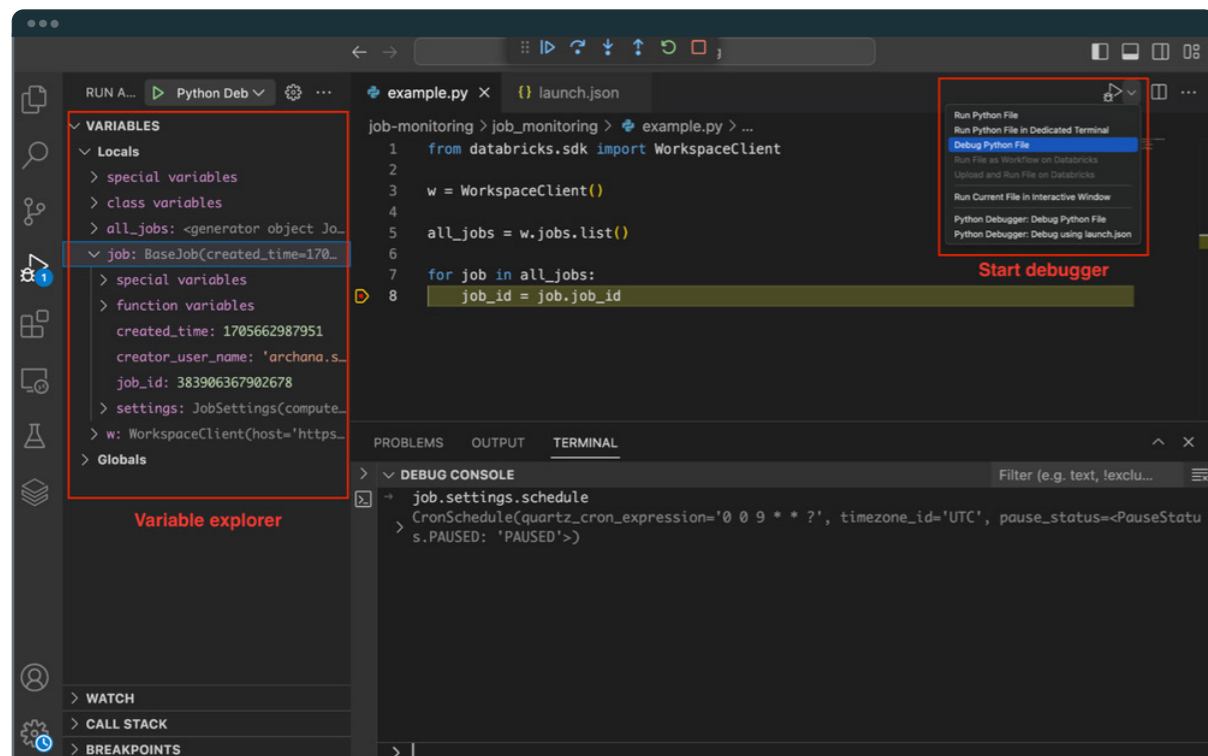


```

1  from dataclasses import dataclass
2  from enum import Enum
3  from typing import Optional
4
5  class CompanyDepartment(Enum):
6      MARKETING = 'MARKETING'
7      SALES = 'SALES'
8      ENGINEERING = 'ENGINEERING'
9
10 @dataclass
11 class Employee:
12     name: str
13     email: str
14     department: Optional[CompanyDepartment] = None
15
16 emp = Employee('Bob', 'bob@company.com', CompanyDepartment.ENGINEERING)
  
```

In the Python SDK all of the data classes, enums and APIs belong to the same module for a service located under **databricks.sdk.service** — e.g., databricks.sdk.service.jobs, databricks.sdk.service.billing, databricks.sdk.service.sql.

For our example, we'll need to loop through all of the jobs and make a decision on whether or not they should be paused. I'll be using a **debugger** in order to look at a few example jobs and get a better understanding of what a 'BaseJob' looks like. The Databricks VS Code extension comes with a debugger that can be used to troubleshoot code issues interactively on Databricks via **Databricks Connect**. But, because I do not need to run my code on a cluster, I'll just be using the standard **Python debugger**. I'll set a breakpoint inside for my loop and use the VS Code Debugger to examine a few examples. A breakpoint allows us to stop code execution and interact with variables during our debugging session. This is preferable over print statements, as you can use the debugging console to interact with the data as well as progress the loop. In this example I'm looking at the **settings** field and drilling down further in the debugging console to take a look at what an example job schedule looks like:



Inspecting BaseJob in the VS Code debugger

We can see a BaseJob has a few top-level attributes and has a more complex Settings type that contains most of the information we care about. At this point, we have our WorkspaceClient and are iterating over the jobs in our workspace. To flag problematic jobs and potentially take some action, we'll need to better understand **job.settings.schedule**. We need to figure out how to programmatically identify if a job has a schedule and flag if it's not paused. For this we'll be using another handy utility for code navigation — **Go to Definition**. I've opted to Open Definition to the Side (⌘ F12) in order to reduce switching to a new window. This will allow us to quickly navigate through the data class definitions without having to switch to a new window or exit our IDE:

As we can see, a BaseJob contains some top-level fields that are common among jobs such as 'job_id' or 'created_time'. A job can also have various settings (JobSettings). These configurations often differ between jobs and encompass aspects like notification settings, tasks, tags and the schedule. We'll be focusing on the schedule field, which is represented by the CronSchedule data class. CronSchedule contains information about the pause status (PauseStatus) of a job. PauseStatus in the SDK is represented as an enum with two possible values — **PAUSED** and **UNPAUSED**.

Tip: VSCode + Pylance provides code suggestions, and you can enable auto imports in your **User Settings or on a per-project basis in Workspace Settings**. By default, only top-level symbols are suggested for auto import and suggested code (see [original GitHub issue](#)). However, the SDK has nested elements we want to generate suggestions for. We actually need to go down 5 levels — databricks.sdk.service.jobs.<Enum|Dataclass>. In order to take full advantage of these features for the SDK, I added a couple of workspace settings:

- *Selection of the VSCode Workspace settings.json*

```

1  ...
2  "python.analysis.autoImportCompletions": true,
3  "python.analysis.indexing": true,
4  "python.analysis.packageIndexDepths": [
5  {
6      "name": "databricks",
7      "depth": 5,
8      "includeAllSymbols": true
9  }
10 ]
11 ...

```

Putting it all together:

I broke out the policy logic into its own function for unit testing, added some logging and expanded the example to check for any jobs tagged as an exception to our policy. Now we have:

- *Logging out of policy jobs*

```

1  import logging
2
3  from databricks.sdk import WorkspaceClient
4  from databricks.sdk.service.jobs import CronSchedule, JobSettings, PauseStatus
5
6  # Initialize WorkspaceClient
7  w = WorkspaceClient()
8
9  def update_new_settings(job_id, quartz_cron_expression, timezone_id):
10     """Update out of policy job schedules to be paused"""
11     new_schedule = CronSchedule(
12         quartz_cron_expression=quartz_cron_expression,
13         timezone_id=timezone_id,
14         pause_status=PauseStatus.PAUSED,
15     )
16     new_settings = JobSettings(schedule=new_schedule)
17
18     logging.info(f"Job id: {job_id}, new_settings: {new_settings}")
19     w.jobs.update(job_id, new_settings=new_settings)
20
21 def out_of_policy(job_settings: JobSettings):
22     """Check if a job is out of policy.
23     If it unpaused and has a schedule and is not tagged as keep_alive
24     Return true if out of policy, false if in policy
25     """
26
27     tagged = bool(job_settings.tags)
28     proper_tags = tagged and "keep_alive" in job_settings.tags
29     paused = job_settings.schedule.pause_status is PauseStatus.PAUSED
30
31     return not paused and not proper_tags
32
33 all_jobs = w.jobs.list()
34 for job in all_jobs:
35     job_id = job.job_id
36     if job.settings.schedule and out_of_policy(job.settings):
37         schedule = job.settings.schedule
38
39         logging.info(
40             f"Job name: {job.settings.name}, Job id: {job_id}, creator: {job.creator_
41             user_name}, schedule: {schedule}"
42         )
43     ....

```

Now we have not only a working example but also a great foundation for building out a generalized job monitoring tool. We're successfully connecting to our workspace, listing all the jobs and analyzing their settings, and, when we're ready, we can simply call our `update_new_settings` function to apply the new paused schedule. It's fairly straightforward to expand this to meet other requirements you may want to set for a workspace — for example, swap job owners to service principles, add tags, edit notifications or audit job permissions. See the example in the [GitHub repository](#).

SCHEDULING A JOB ON DATABRICKS

You can run your script anywhere, but you may want to schedule scripts that use the SDK to run as a Databricks Workflow or job on a small single-node cluster. When running a Python notebook interactively or via automated workflow, you can take advantage of default Databricks Notebook authentication. If you're working with the Databricks WorkspaceClient and your cluster meets the [requirements listed in the docs](#), you can initialize your WorkspaceClient without needing to specify any other configuration options or environment variables — it works automatically out of the box.

The screenshot displays the Databricks job monitoring interface. The top navigation bar shows the path: Workflows > Jobs > job-monitoring-example > Run 399648124376431. The job status is 'Original (latest) - Succeeded'. The main panel shows the job output, which includes a list of jobs and their settings. The output text is as follows:

```
proper_tags = tagged and keep_alive in job_settings.tags
paused = job_settings.schedule.pause_status is PauseStatus.PAUSED

return not paused and not proper_tags

all_jobs = w.jobs.list()
for job in all_jobs:
    job_id = job.job_id
    if job.settings.schedule and out_of_policy(job.settings):
        schedule = job.settings.schedule

        print(f"Job name: {job.settings.name}, Job id: {job_id}, schedule: {schedule.quartz_cron_expression}")

Job name: test-mlops-project-natya-model-training-job, Job id: 512654776773878, schedule: 0 0 9 * * ?
Job name: test-mlops-project-natya-write-feature-table-job, Job id: 581617178734662, schedule: 0 0 7 * * ?
Job name: test-mlops-project-natya-batch-inference-job, Job id: 661538788172146, schedule: 0 0 11 * * ?
Job name: staging-mlops-project-natya-write-feature-table-job, Job id: 980297457859299, schedule: 0 0 7 * * ?
Job name: staging-mlops-project-natya-model-training-job, Job id: 574748586272652, schedule: 0 0 9 * * ?
Job name: staging-mlops-project-natya-batch-inference-job, Job id: 848787196780542, schedule: 0 0 11 * * ?
Job name: test-mlops-project-stijn-batch-inference-job, Job id: 3381319250929, schedule: 0 0 11 * * ?
Job name: test-mlops-project-stijn-write-feature-table-job, Job id: 182963295384967, schedule: 0 0 7 * * ?
Job name: test-mlops-project-stijn-model-training-job, Job id: 315725851631785, schedule: 0 0 9 * * ?
Job name: staging-mlops-project-stijn-batch-inference-job, Job id: 613191086293224, schedule: 0 0 11 * * ?
Job name: staging-mlops-project-stijn-model-training-job, Job id: 572588951767651, schedule: 0 0 9 * * ?
Job name: staging-mlops-project-stijn-write-feature-table-job, Job id: 563143323231114, schedule: 0 0 7 * * ?
Job name: SAT Driver Notebook, Job id: 741617137828625, schedule: 0 0 8 ? * Mon,Wed,Fri
Job name: Multi_Hop, Job id: 1060544822804933, schedule: 18 57 11 * * ?
```

The right sidebar shows the 'Task run details' for the job. The job ID is 640112777346392. The job run ID is 399648124376431. The task run ID is 877518783773277. The job was run as 'kmahoney-sdk' and was launched manually. It started on 01/22/2024 at 09:55:33 PM and ended on 01/22/2024 at 09:58:18 PM. The duration was 2m 45s. The status is 'Succeeded'. The queue duration is 0. The lineage information is 'No lineage information for this job. Learn more'. The notebook is located at [/Users/kimberly.mahoney@databricks.com/job_monitoring](#). The compute configuration is 'Job_cluster' with a driver of 'i3.xlarge' and workers of 'i3.xlarge' (1 worker). The cluster is on-demand and Spot, falling back to On-demand - T3.3 LTS Photon (includes Apache Spark 3.4.1, Scala 2.12) - us-east-1b. The dependent libraries are 'databricks-sdk==0.17.0 (PyPI)'.

CONCLUSION

In conclusion, the Databricks SDKs offer limitless potential for a variety of applications. We saw how the Databricks SDK for Python can be used to automate a simple yet crucial maintenance task and also saw an example of an OSS project that uses the Python SDK to integrate with the Databricks Platform. Regardless of the application you want to build, the SDKs streamline development for the Databricks Platform and allow you to focus on your particular use case. The key to quickly mastering a new SDK such as the Databricks Python SDK is setting up a proper development environment. Developing in an IDE allows you to take advantage of features such as a debugger, parameter info and code completion, so you can quickly navigate and familiarize yourself with the codebase. Visual Studio Code is a great choice for this as it provides the above capabilities and you can utilize the VSCode extension for Databricks to benefit from unified authentication.

Any feedback is greatly appreciated and welcome. Please raise any issues in the [Python SDK GitHub repository](#). Happy developing!

ADDITIONAL RESOURCES

- [Databricks SDK for Python Documentation](#)
- DAIS Presentation: [Unlocking the Power of Databricks SDKs](#)
- How to install Python libraries in your local development environment: [How to Create and Use Virtual Environments in Python With Poetry](#)
- [Installing the Databricks Extension for Visual Studio Code](#)



03

Ready-to-Use Notebooks and Datasets

This section includes several Solution Accelerators — free, ready-to-use examples of data solutions from different industries ranging from retail to manufacturing and healthcare. Each of the following scenarios includes notebooks with code and step-by-step instructions to help you get started. Get hands-on experience with the Databricks Data Intelligence Platform by trying the following for yourself:



Overall Equipment Effectiveness

Ingest equipment sensor data for metric generation and data-driven decision-making

[Explore the Solution](#)



Real-Time Point-of-Sale Analytics

Calculate current inventories for various products across multiple store locations with Delta Live Tables

[Explore the Solution](#)



Digital Twins

Leverage digital twins — virtual representations of devices and objects — to optimize operations and gain insights

[Explore the Solution](#)



Recommendation Engines for Personalization

Improve customers' user experience and conversion with personalized recommendations

[Explore the Solution](#)



Understanding Price Transparency Data

Efficiently ingest large healthcare datasets to create price transparency for better understanding of healthcare costs

[Explore the Solution](#)

Additional Solution Accelerators with ready-to-use notebooks can be found here:

[Databricks Solution Accelerators](#)

04

Case Studies

COX AUTOMOTIVE



Cox Automotive — changing the way the world buys, sells and uses vehicles

“We use Databricks Workflows as our default orchestration tool to perform ETL and enable automation for about 300 jobs, of which approximately 120 are scheduled to run regularly.”

— Robert Hamlet, Lead Data Engineer, Enterprise Data Services, Cox Automotive

INDUSTRY

Automotive

SOLUTION

Data-Driven ESG, Customer Entity Resolution, Demand Forecasting, Product Matching

PLATFORM

Workflows, Unity Catalog, Delta Sharing, ETL

CLOUD

Azure

Cox Automotive Europe is part of Cox Automotive, the world’s largest automotive service organization, and is on a mission to transform the way the world buys, sells, owns and uses vehicles. They work in partnership with automotive manufacturers, fleets and retailers to improve performance and profitability throughout the vehicle lifecycle. Their businesses are organized around their customers’ core needs across vehicle solutions, remarketing, funding, retail and mobility. Their brands in Europe include Manheim, Dealer Auction, NextGear Capital, Modix and Codeweavers.

Cox’s enterprise data services team recently built a platform to consolidate the company’s data and enable their data scientists to create new data-driven products and services more quickly and easily. To enable their small engineering team to unify data and analytics on one platform while enabling orchestration and governance, the enterprise data services team turned to the Databricks Data Intelligence Platform, Workflows, Unity Catalog and Delta Sharing.