Airflow 101

知りたいこと

-
- Airflow
    - What is Hook
    - How to create an operator
    - Import best practice
    - House keeping
        - Logging
    - Unit test in Airflow
    - Redshift Connector

Prerequisite

astro dev init
astro dev start
localhost:8080 でadmin, adminでlogin

.airflowignore
SQLなど無視したいfile

Airflow Architecture

Must have arguments

- start_date
- end_date

- schedule_interval
    - use cron expressions as much as possible.
    - cron vs timedelta
        - cron is stateless, but **timedelta is stateful**.
        - ex) Suppose you want to run it daily at 0am. If your schedule_interval=timedelta(days = 1) and start_date = 2024-01-01 10am, then you run at 10 am. So you can't run it at 0 am.
- dagrun_timeout = timedelta(minutes=60)
- tags=[]
- catchup = False
    - in general False as best practice because if your start_date happened to be a year ago, then you run unnecessary runs. you can still backfill manually.
- max_active_run = 1
    -

Core Concepts

Task idempotent

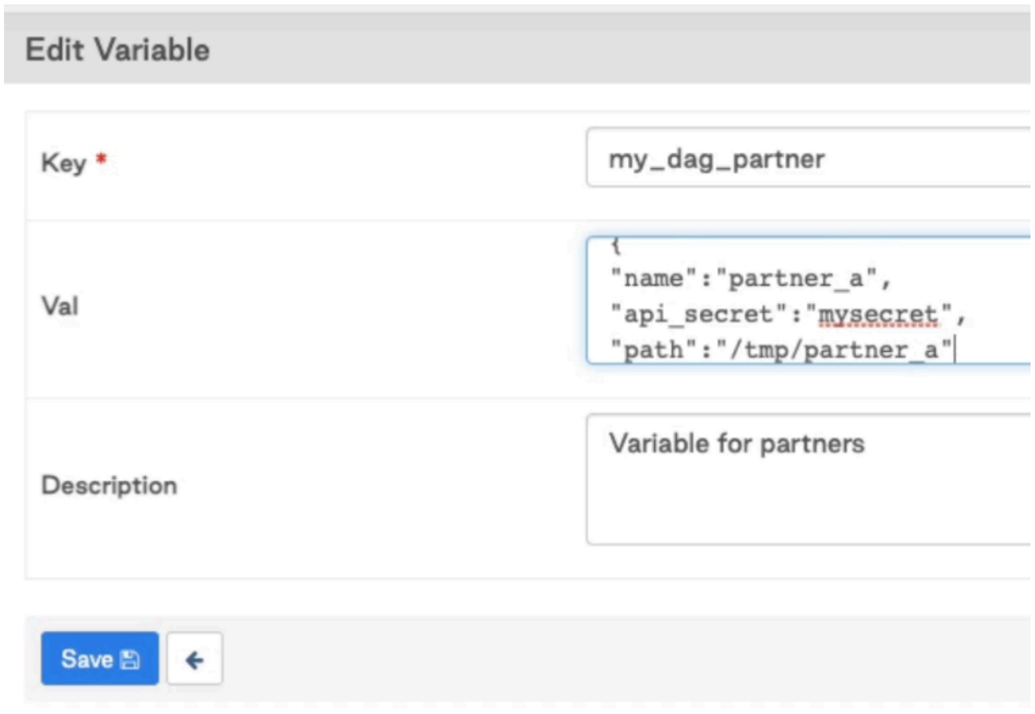
ex) create table xxx is NOT idempotent

Variable / ENV

The goal of the variable is to reuse objects again and again such as connections. This way, you never store your credential in the code.

step1: define your variable as ENV in the UI.
Note: by naming it as xxx**_secret** , you can hide your credential!!s
step2: from airflow.models import Variable
step3: use it by Variable.get("my_variable")

<u>**Tips**</u>
You can define variables in python dict. The benefit of doing this is that you only need to run Variable.get() one time.
ex) credentails_dict = Variable.get("my_variable_secret", deserialize_json = True)


Edit Variable

Key *	my_dag_partner
Val	{ "name":"partner_a", "api_secret":"mysecret", "path":"/tmp/partner_a"
Description	Variable for partners

Callout!!

- If you access Variable.get(“my_variable”) outside of task, connection is established when a DAG is parsed, even if DAG is not running yet. This impacts heavy read on the prod DB.
-

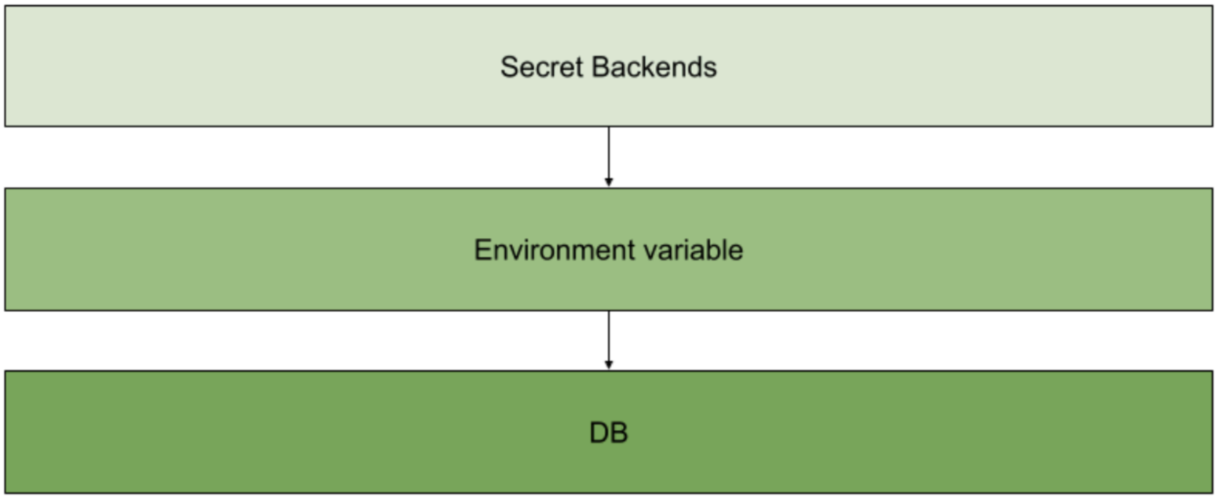
ENV variable

In dockerfile, define your ENV as follows.

```
ENV AIRFLOW_VAR_xxx = ‘{“name”: “”, …}’
```

Note:
If you define your ENV variable, you have to remove it from the UI.
Also, you won’t be able to see ENV var from the UI anymore.

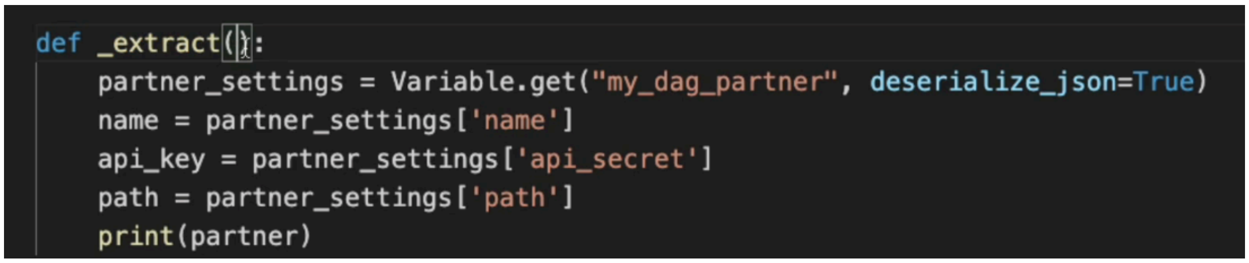
ɜ: Finally, here is the order in which Airflow checks if your variable/connection exists:



Question: which DB table name stores ENV variables ?

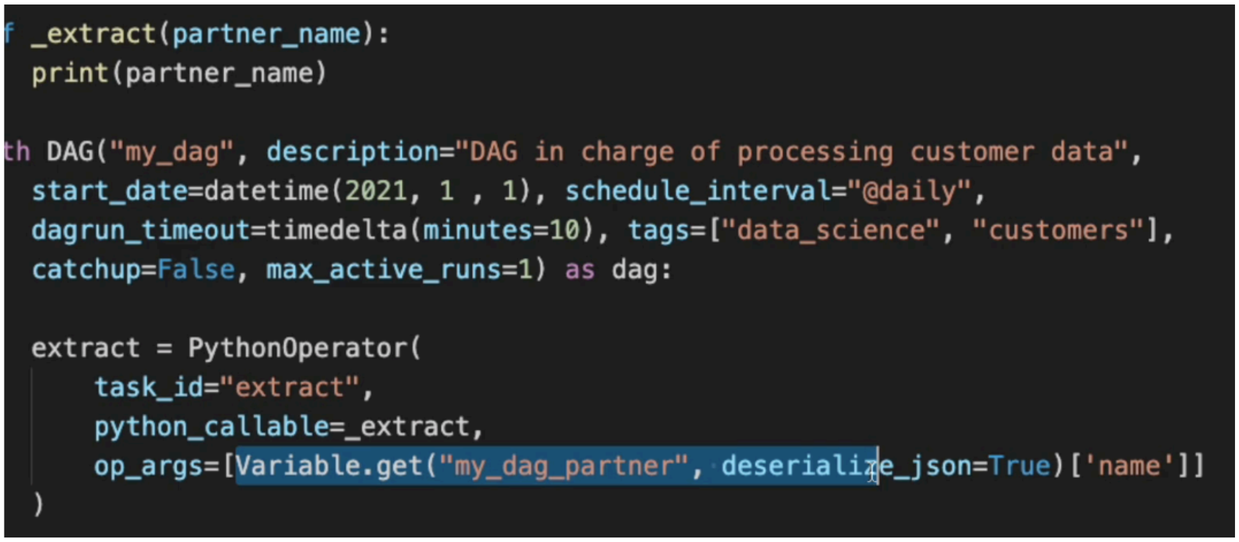
python_callable function

Suppose you want to pass a name arg to this function.
before)



```
def _extract():
    partner_settings = Variable.get("my_dag_partner", deserialize_json=True)
    name = partner_settings['name']
    api_key = partner_settings['api_secret']
    path = partner_settings['path']
    print(partner)
```

After) you can pass parameter partner_name by using op_args
Here, you pass name from Variable in the UI.

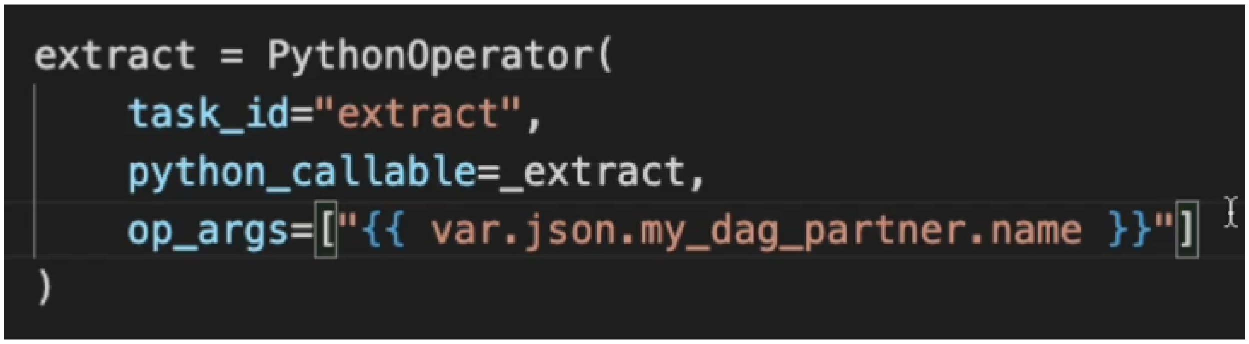


```
f _extract(partner_name):
  print(partner_name)

th DAG("my_dag", description="DAG in charge of processing customer data",
  start_date=datetime(2021, 1 , 1), schedule_interval="@daily",
  dagrun_timeout=timedelta(minutes=10), tags=["data_science", "customers"],
  catchup=False, max_active_runs=1) as dag:

  extract = PythonOperator(
      task_id="extract",
      python_callable=_extract,
      op_args=[Variable.get("my_dag_partner", deserialize_json=True)['name']]
  )
```

Alternatively, you can also pass variable by var.json.variable_name.json_key



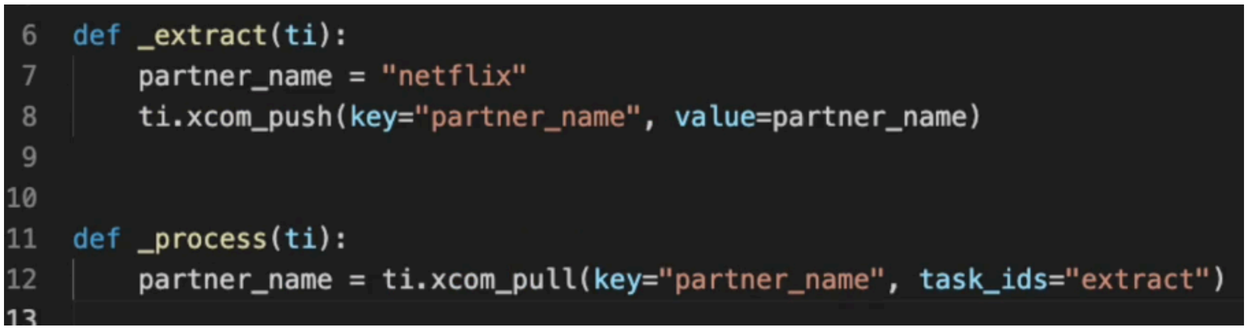
```
extract = PythonOperator(
    task_id="extract",
    python_callable=_extract,
    op_args=["{{ var.json.my_dag_partner.name }}"]
)
```

op_args vs kwargs

xxx

XCOM

If your variable can be stored as Variable in UI, do that.



```python
def _extract(ti):
    partner_name = "netflix"
    ti.xcom_push(key="partner_name", value=partner_name)


def _process(ti):
    partner_name = ti.xcom_pull(key="partner_name", task_ids="extract")
```

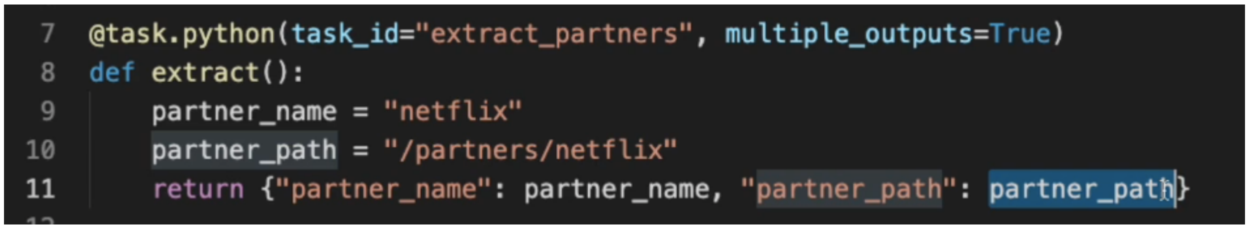
Limitation of XCOM

variable size is limited to 2 GB for sqlite, 1 GB size for postgres, 64 KB for mysql.

How can I create a separate XCOMs in one json but without pushing twice ??

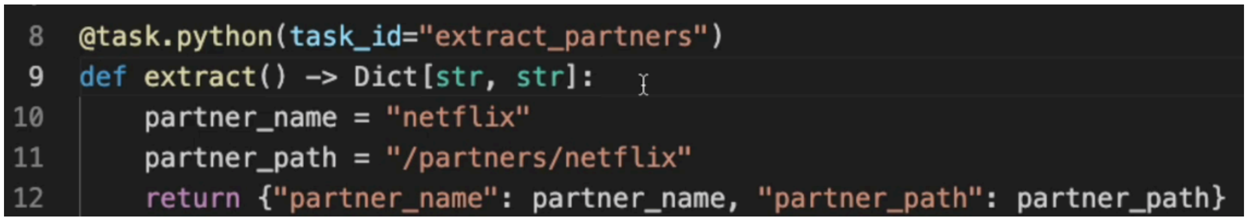
Option1 : Use task API!

@task.python(task_id = “xx”, **multiple_outputs = True**)



```python
@task.python(task_id="extract_partners", multiple_outputs=True)
def extract():
    partner_name = "netflix"
    partner_path = "/partners/netflix"
    return {"partner_name": partner_name, "partner_path": partner_path}
```

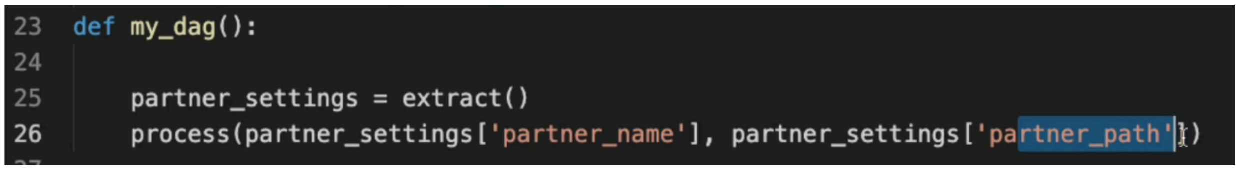
Option2 : Use Dict[str, str]



```python
@task.python(task_id="extract_partners")
def extract() -> Dict[str, str]:
    partner_name = "netflix"
    partner_path = "/partners/netflix"
    return {"partner_name": partner_name, "partner_path": partner_path}
```

So, how do I consume ?

Just access python dict for each variable.



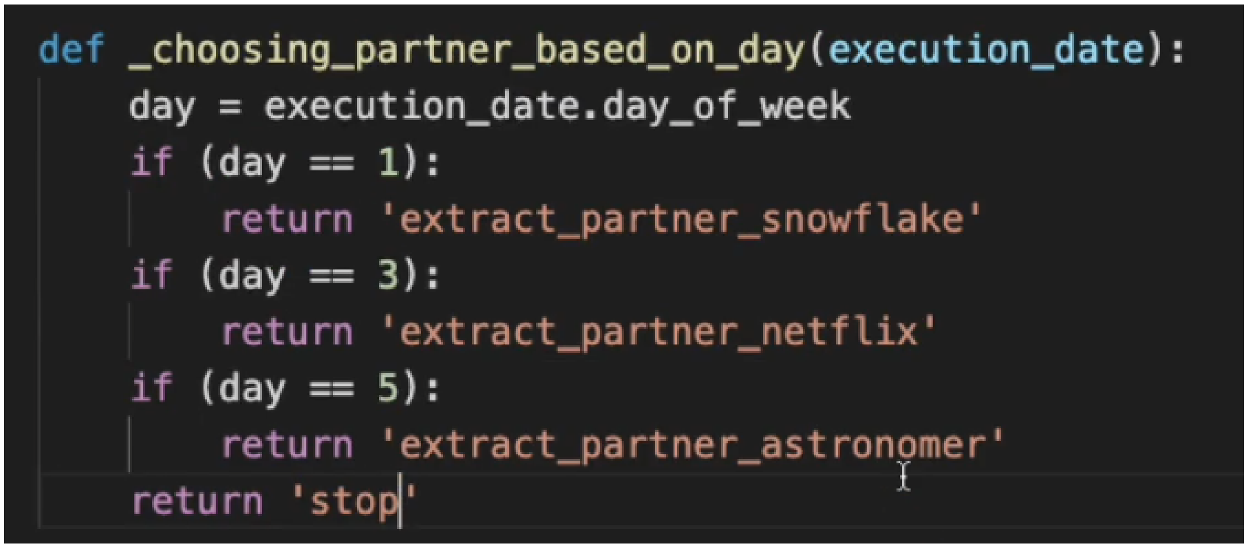
```python
def my_dag():

    partner_settings = extract()
    process(partner_settings['partner_name'], partner_settings['partner_path'])
```

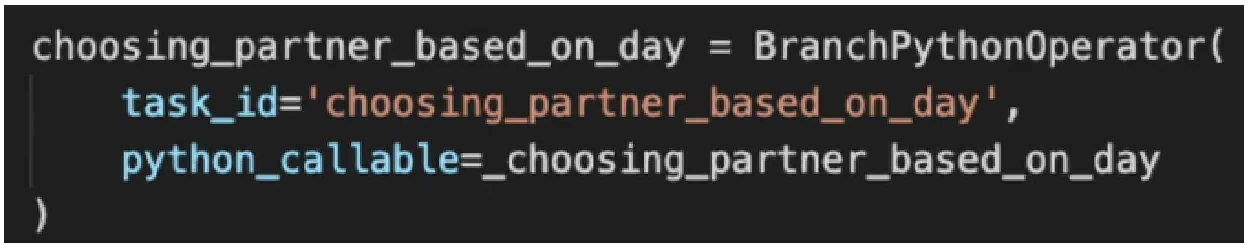
SubDAG vs TaskGroup

SubDAG is a sensor in reality. It waits before moving to the next task.

Python Branch Operator

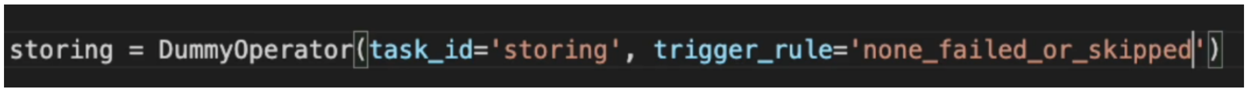


```python
def _choosing_partner_based_on_day(execution_date):
    day = execution_date.day_of_week
    if (day == 1):
        return 'extract_partner_snowflake'
    if (day == 3):
        return 'extract_partner_netflix'
    if (day == 5):
        return 'extract_partner_astronomer'
    return 'stop'
```



```python
choosing_partner_based_on_day = BranchPythonOperator(
    task_id='choosing_partner_based_on_day',
    python_callable=_choosing_partner_based_on_day
)
```

ってか、trigger_rule にnone_failed_or_skippedなんてあったっけ？



```python
storing = DummyOperator(task_id='storing', trigger_rule='none_failed_or_skipped')
```

気をつけて、これ、all parents have not failed (`failed` or `upstream_failed`) and at least **one parent has succeeded**を含んでる！！
https://airflow.apache.org/docs/apache-airflow/1.10.10/concepts.html#:~:text=none_failed_or_skipped%20%3A%20all%20parents%20have%20not,for%20show%2C%20trigger%20at%20will

Trigger rule

What is the difference b/w none_skipped and all done?
In Airflow, the definition of done includes the “skipped” state !!!
so a task with a trigger rule set to `all_done` will execute regardless of the outcome of its upstream tasks. It will run if all upstream tasks are in a terminal state (success, failure, or **skipped**).

Depends on Past

In general, I don’t like this config because the DAG because stateful.
if depends_on_past = True then the next DAG won’t be triggered until the previous run is cleared.

Dependencies

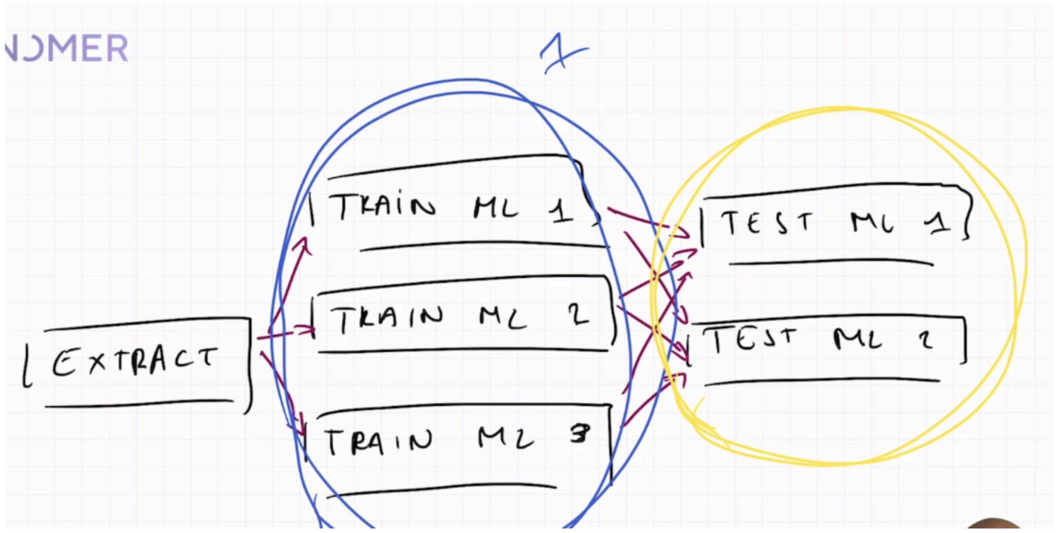
In airflow dependencies, you can’t do this.
[task_1,task_2, task_3] >> [task_4, task_5, task_6]
but this is repetitive.
[task_1,task_2, task_3] >> task_4
[task_1,task_2, task_3] >> task_5
[task_1,task_2, task_3] >> task_6

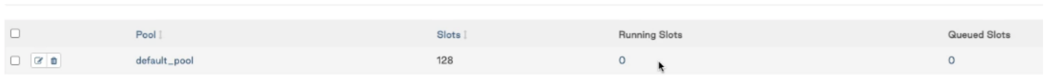
Alternative solution 1:
from airflow.models.baseoperator import cross_downstream
cross_downstream([task_1,task_2, task_3], [task_4, task_5, task_6])

Alternative solution 2:
task_group >> [task_4, task_5, task_6]

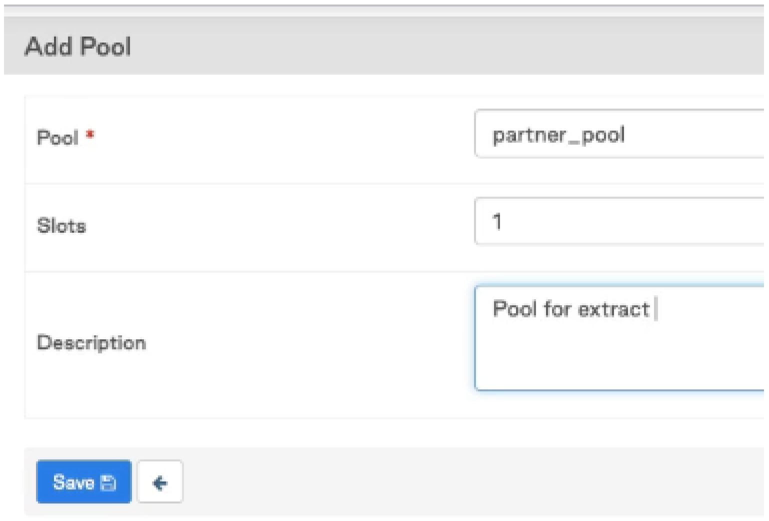
Concurrency

- task level concurrency is called “concurrency”. This can be defined at the DAG level config. Default max is 32
    - You can also control task level concurrency by “task_concurrency=1”
    - このtaskだけ並行処理しないで! みたいな
- DAG level concurrency can be controlled by “max_active_run”. Default max is 16. If you want to avoid any edge case, then set this to 1.
- **pool**:

- I want to set concurrency 1 for the blue tasks, but want to run yellow tasks in parallel.
- 

- each pool is allocated slots.
- 

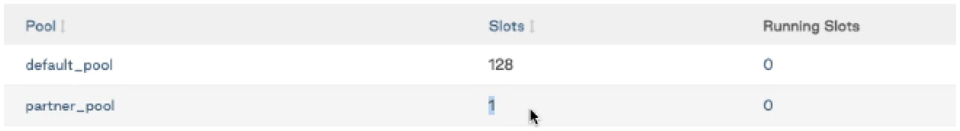
	Pool	Slots	Running Slots	Queued Slots
	default_pool	128	0	0

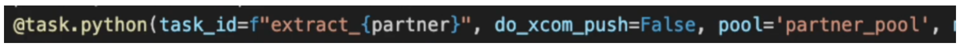
- So when you define a task, you can use different pool:
  - 

    Add Pool

Pool *	partner_pool
Slots	1
Description	Pool for extract

  -
  - so here you see two different pool.

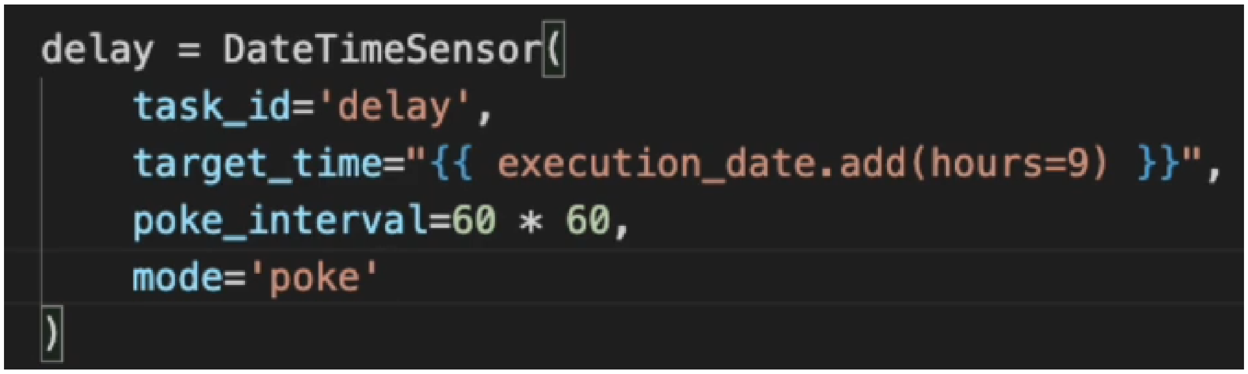
    

Pool	Slots	Running Slots
default_pool	128	0
partner_pool	1	0
  -
  - 

    ```
    @task.python(task_id=f"extract_{partner}", do_xcom_push=False, pool='partner_pool',
    ```
  - call a pool with only slot = 1
-

Sensors

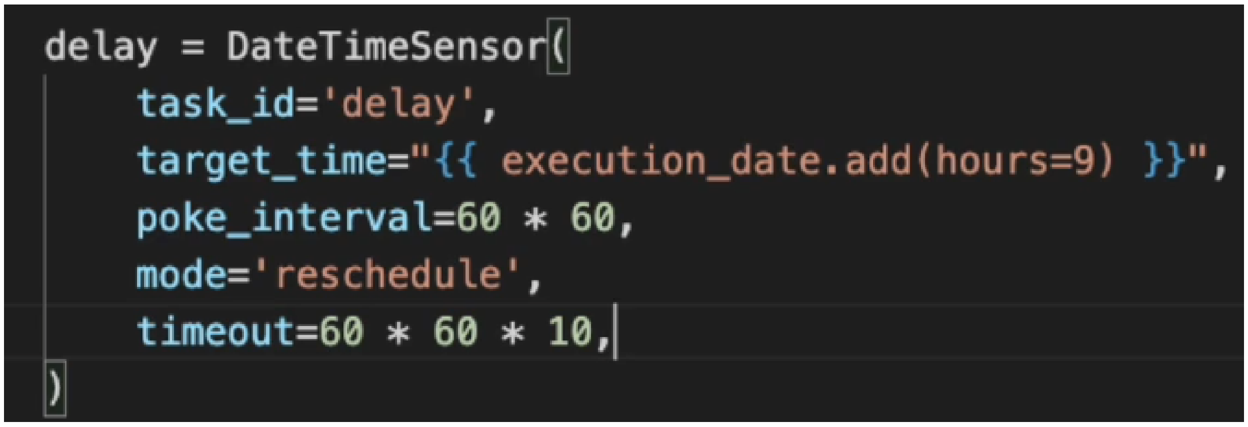
case 1: you want to add a delay



```
delay = DateTimeSensor(
    task_id='delay',
    target_time="{{ execution_date.add(hours=9) }}",
    poke_interval=60 * 60,
    mode='poke'
)
```

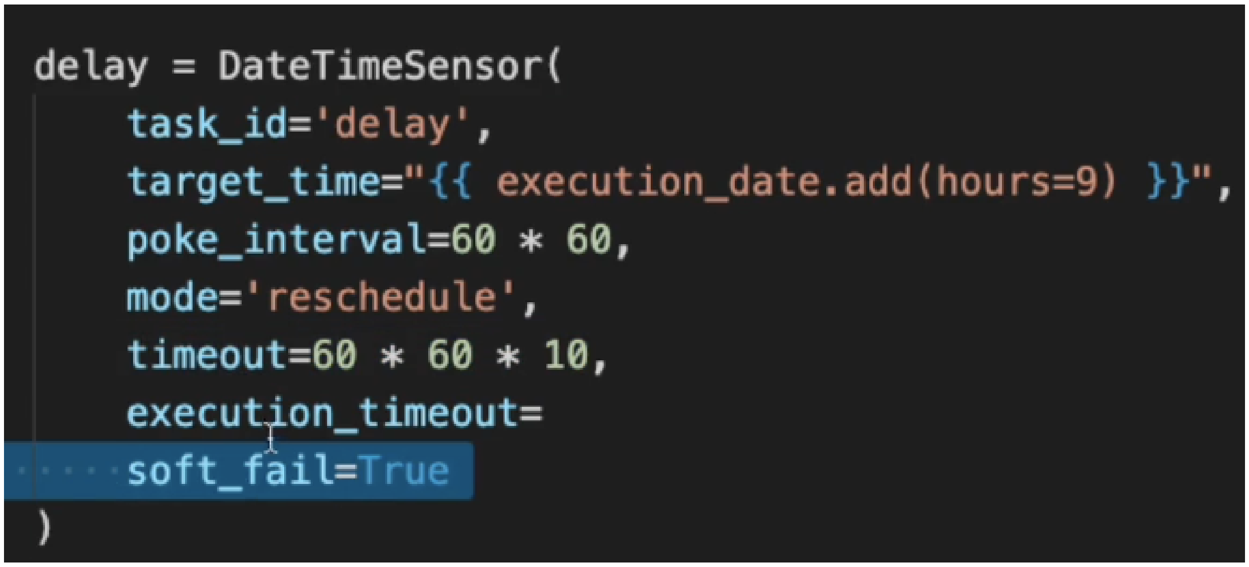
what is mode?
poke will check if a task is done every poke_interval. poke_interval unit is in seconds by default.

Important:
make sure you set **timeout** in your sensor



```
delay = DateTimeSensor(
    task_id='delay',
    target_time="{{ execution_date.add(hours=9) }}",
    poke_interval=60 * 60,
    mode='reschedule',
    timeout=60 * 60 * 10,
)
```

Tips: if you use timeout in the sensor, use soft_fail = True together !!
video:https://academy.astronomer.io/astronomer-certification-apache-airflow-dag-authoring-preparation/885242
if soft_fail = True, as soon as timer times out, the sensor will be skipped instead of fail.


```
delay = DateTimeSensor(
    task_id='delay',
    target_time="{{ execution_date.add(hours=9) }}",
    poke_interval=60 * 60,
    mode='reschedule',
    timeout=60 * 60 * 10,
    execution_timeout=
    soft_fail=True
)
```

<u>what is the difference between timeout and execution_timeout which is from base operator's ?</u>

- execution_timeout has no default value
- execution_timeout is a hard stop. It does not work with soft_fail.
    - so timeout with soft_fail = True will be skipped, but execution_timeout will fail.

Usually I prefer execution_timout to set the task to fail, unless there is no downstream request to set it to skip.

DAG run timeout & task execution_timeout

This is a must set DAG.
Now the question is, should you always set execution_timeout to every task?

Usually Yes, because
- **Resource Management:** It helps in releasing system resources faster by failing tasks that are running longer than they should.
- **Faster Failures:** If a task takes much longer than expected, setting a task-level timeout will fail it quickly, rather than waiting for the DAG-level timeout.
- **Debugging:** It's easier to debug long-running tasks if they have specific timeouts, as this isolates the slow task rather than looking at the whole DAG.

When you might not need to set it on every task:

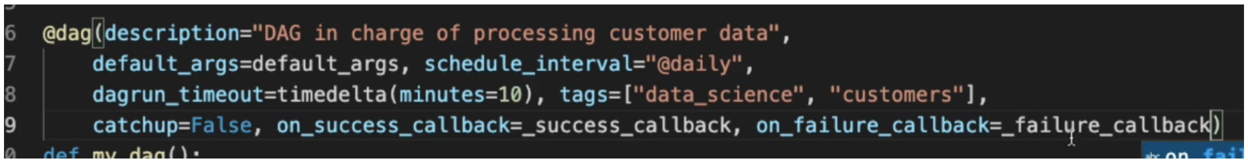
- If all tasks in the DAG have similar execution characteristics and durations, a DAG-level `dag_run_timeout` might be sufficient.

I want to handle a failure

option1: create a downstream task with trigger_rule = TriggerRule.FAILURE

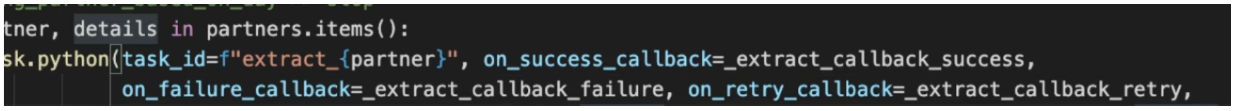
option2: create a callback function

DAG level



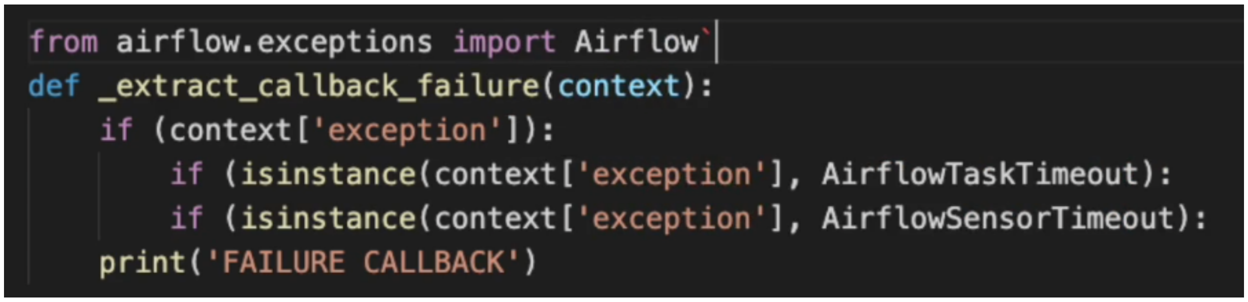
```
@dag(description="DAG in charge of processing customer data",
    default_args=default_args, schedule_interval="@daily",
    dagrun_timeout=timedelta(minutes=10), tags=["data_science", "customers"],
    catchup=False, on_success_callback=_success_callback, on_failure_callback=_failure_callback)
def my_dag():
```

Task level callback



```
tner, details in partners.items():
sk.python(task_id=f"extract_{partner}", on_success_callback=_extract_callback_success,
           on_failure_callback=_extract_callback_failure, on_retry_callback=_extract_callback_retry,
```

すごいこと
You can split the call back logic based on Exception types.

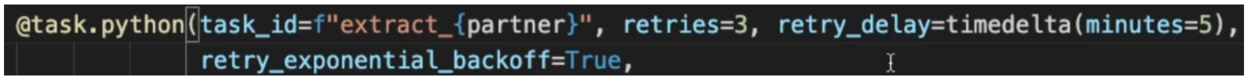


```
from airflow.exceptions import Airflow`
def _extract_callback_failure(context):
    if (context['exception']):
        if (isinstance(context['exception'], AirflowTaskTimeout):
        if (isinstance(context['exception'], AirflowSensorTimeout):
    print('FAILURE CALLBACK')
```

Retry

if you use retry, maybe setting retry_delay=timedelta(minutes=1)
might help.

もっといいのは
retry_exponential_backoff =True
で、これ使うなら
max_retry_delay=timedelta(minutes=10)
をちゃんとsetして。



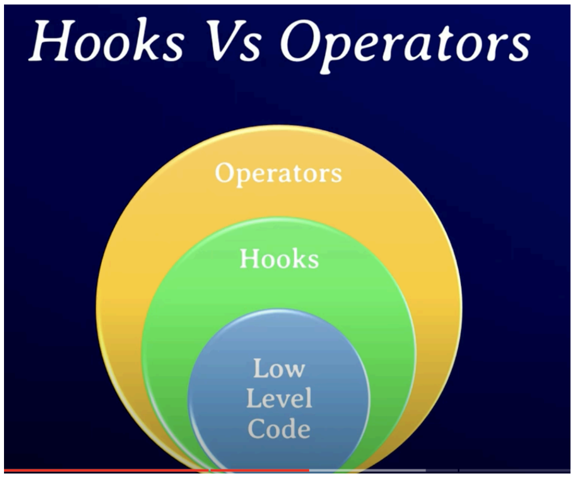
```
@task.python(task_id=f"extract_{partner}", retries=3, retry_delay=timedelta(minutes=5),
             retry_exponential_backoff=True,
```

SLA vs dag_run_timout

- Goal of SLA is to only notify. It does not fail the DAG.
    - sla_miss_callback is a DAG arg.
    - **IMPORTANT: if you trigger your DAG manually, then SLA is not checked.**
    - **Also remember that the clock starts from the scheduled execution_date, not the calendar time that the DAG is executed.**
        - ex) if execution_date is midnight 12 am and SLA is 24 hours then notification happens in the 24 am UTC, but in EST it's actually 8 pm People often think that the clock starts from calendar.now() when execution starts.
- Goal of dag_run_timout is to fail the DAG.
-

三菱送金

Hook



Purpose of Airflow Hooks:

- **Connection Management**: Hooks manage the connection to external services such as databases, message queues, or cloud storage by leveraging Airflow's built-in connection management.
- **Code Reusability**: Hooks provide reusable code for interacting with external systems, reducing redundancy in Operators and DAGs.
- **Abstraction**: They abstract the logic required to connect to external services, simplifying task code and reducing complexity in DAGs.

- **Standardization**: Hooks standardize the way Airflow interacts with external systems, ensuring uniformity and ease of debugging.

Common Hook Types in Airflow:

- **Database Hooks**: Connect to databases (e.g., `PostgresHook`, `MySqlHook`, `JdbcHook`).
- **Cloud Hooks**: Interact with cloud services (e.g., `GCSHook` for Google Cloud Storage, `S3Hook` for AWS S3).
- **API Hooks**: Interact with REST APIs or other web services (e.g., `HttpHook`).

ex1) PostgresHook

```
from airflow.hooks.postgres_hook import PostgresHook
from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults

class MyPostgresOperator(BaseOperator):

    @apply_defaults
    def __init__(self, *args, **kwargs):
        super(MyPostgresOperator, self).__init__(*args, **kwargs)

    def execute(self, context):
        # Initialize the PostgresHook
        postgres_hook = PostgresHook(postgres_conn_id='my_postgres_conn')

        # Run a SQL query
        sql = "SELECT COUNT(*) FROM my_table"
        result = postgres_hook.get_first(sql)
        self.log.info(f"Count from my_table: {result[0]}")
```

Hook Best practice | re-initiate hook every task

From Aifrlow best practice, is it actually recommended to re-initiate postgres_hook every task?
ex) postgres_hook = PostgresHook(postgres_conn_id='my_postgres_conn')

From an **Airflow best practice** perspective, it is actually **recommended** to re-initialize the `PostgresHook` (or any hook) within each task. The reasoning behind this practice comes down to how Airflow is designed and the principles of task isolation and idempotency.

Why Reinitializing Hooks in Each Task is Recommended:

1. **Task Isolation**:
    - Airflow tasks are designed to run in isolation, often in separate worker processes or containers. Re-initializing a hook in each task ensures that each task runs independently, without relying on shared resources that could lead to side effects or race conditions.
    - If you reuse a hook across tasks, you risk introducing dependencies between tasks that break the isolation, making the DAG harder to debug and maintain.
2. **Idempotency**:
    - Airflow tasks are meant to be **idempotent**, meaning that they can be safely retried without causing unintended side effects. Reinitializing a hook for each task ensures that if a task fails and retries, it can start fresh without relying on a previously established connection or state.
    - Reusing a connection object between tasks can lead to issues if, for instance, the connection is dropped or timed out between task executions.
3. **Connection Management**:
    - When you reinitialize a hook, it also re-establishes a connection to the external system (like a database). In many cases, maintaining long-lived connections across tasks can lead to timeouts or stale connections, especially if the tasks are distributed across different workers.
    - Airflow manages connections via its `Connections` feature, and it's optimized for re-initializing connections efficiently. Re-initializing the hook ensures that each task gets a fresh, valid connection from the pool.
4. **Modularity and Reusability**:
    - Hooks are designed to be lightweight and stateless. Reinitializing the hook in each task allows for better modularity, meaning each task can be more easily understood, reused, or modified without worrying about the internal state of the hook from a previous task.
    - Airflow encourages defining tasks in a way that they can be reused in other DAGs, and having the hook instantiated within the task helps make tasks portable and self-contained.

Key Considerations:

1. **Efficiency**:
    - While reinitializing a hook may seem inefficient, in practice, it’s usually not an issue. Database connections are pooled and cached by the underlying libraries and databases, so the overhead is minimal. Also, Airflow is optimized to handle such reconnections.
    - The cost of initializing a `PostgresHook` (or any other hook) is typically small compared to the cost of the actual work (e.g., running a SQL query).
2. **Scalability**:

  - Airflow scales horizontally by distributing tasks across multiple workers. Re-initializing the hook ensures that each worker can independently handle its tasks without having to share a global state.
  - For instance, in large-scale deployments with distributed workers, sharing the same connection object across tasks may not be feasible or scalable.
3. **Error Handling and Retries**:
   - Reinitializing the hook in each task allows for better error handling and retries. If a database connection fails or a hook encounters an error, reinitializing the hook on task retry ensures a fresh connection, improving resilience.
4. **Session Management**:
   - Long-lived sessions with external systems (e.g., databases) can be problematic. Re-initializing the `PostgresHook` ensures that each task gets a clean session, avoiding issues with session timeouts, stale data, or session state.

Recommended Airflow Practice: Re-initialize Hooks in Each Task

Example:

python

Copy code

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.hooks.postgres_hook import PostgresHook
from airflow.utils.dates import days_ago

def execute_sql(query):
    # Re-initialize the hook in each task
    postgres_hook = PostgresHook(postgres_conn_id='my_postgres_conn')
    conn = postgres_hook.get_conn()
    cursor = conn.cursor()

    cursor.execute(query)
    result = cursor.fetchone()

    # Log or return the result
    return result[0]

# List of SQL queries
sql_queries = [
    "SELECT COUNT(*) FROM table1",
    "SELECT COUNT(*) FROM table2",
    "SELECT COUNT(*) FROM table3"
```

```python
]

# Define the DAG
default_args = {
    'owner': 'airflow',
    'start_date': days_ago(1),
}

with DAG(dag_id='best_practice_dag', default_args=default_args,
schedule_interval=None) as dag:

    # Create a task for each SQL query
    for i, query in enumerate(sql_queries):
        sql_task = PythonOperator(
            task_id=f'execute_query_{i}',
            python_callable=execute_sql,
            op_args=[query]
        )
```

Summary: Hooks are designed to be lightweight and stateless.

- **Best Practice**: Re-initialize hooks in every task to maintain task isolation, ensure idempotency, and handle retries effectively.
- **Minimized Overhead**: The overhead of reinitializing hooks is typically minimal, especially given the benefits of modularity, reliability, and connection pooling.
- **Task Modularity**: Each task is self-contained, making it easier to debug, maintain, and reuse across DAGs.

How to create an operator

これをみなさい Apache Airflow | How to Create an Operator

Import Best practice

Bad Approach:

```
from airflow import DAG

from airflow.decorators import task

import numpy as np  # <-- THIS IS A VERY BAD IDEA! DON'T DO THAT!
```

Good Approach:

```
def print_array():

        """Print Numpy array."""

        import numpy as np  # <- THIS IS HOW NUMPY SHOULD BE IMPORTED IN
THIS CASE

        a = np.arange(15).reshape(3, 5)

        print(a)
```

Housekeeping

Airflow logは定期的に削除しないとfile systemがfullでDAGが動かなくなる。
https://www.brainpad.co.jp/doors/contents/01_tech_2022-12-20-160153/

Redshift connector

RedshiftSQLOperator vs PythonOperator w/ psycopg2

RedshiftSQLOperator is more Airflow managed operator, so it has advantages of

- You don't need to control connection & credential handling
- RedshiftSQLOperator lets you write jinja templates.
    - => You don't need to render your template.

So, if you logic is not complex, generally recommended to use RedshiftSQLOperator.

In a rare situation sometimes we might prefer using psycopg2 (PythonOperator) when:

- You might need to implement additional error handling and logging to ensure that SQL execution issues are captured.
- You want to control session-specific configurations that require manual cursor and connection management.

RedshiftSQLOperatorは、Airflowが管理するオペレーターなので、ロジックが複雑でなければ、一般的にはRedshiftSQLOperatorの使用が推奨される（はず）

利点としては

- 接続や認証の管理をAirflowに任せられる。
- Jinjaテンプレートを使ってSQLを書くことができます。
    - PythonOperatorだとやる必要のある、テンプレートをレンダリングする必要がない。

逆に、稀にpsycopg2（PythonOperator）を使用する方が良い場合があるとしたら、

- SQL実行の問題をキャッチするために、追加のエラーハンドリングやロギングを実装する必要がある場合。
- 手動でカーソルや接続を管理する必要があるセッション固有の設定を制御したい場合。

RedshiftSQLOperator

The `RedshiftSQLOperator` is part of Airflow's newer practice of using more specialized operators that extend the `SQLOperator`. This operator is tailored specifically for executing SQL commands in a Redshift database environment. Here are its key characteristics:

1. Connection Handling: The `RedshiftSQLOperator` uses Airflow's connection framework to manage database connections. This means that you don't need to manually manage database connections or cursors; the operator handles it for you. You only need to specify the connection ID.
2. SQL Templating: This operator supports Airflow's Jinja templating, allowing dynamic injection of parameters into the SQL queries at runtime. This can be particularly useful for tasks that require dynamic SQL generation based on the DAG's execution context or external variables.
3. Integration with Airflow UI: SQL commands and their results are better integrated with the Airflow UI. Errors and logs are more systematically captured and displayed, which aids in debugging and monitoring.
4. Ease of Use: Generally, it provides a cleaner and more declarative approach to defining tasks that involve SQL operations on Redshift. This aligns with Airflow's design philosophy of defining workflows declaratively rather than imperatively.

RedshiftOperator (or using psycopg2 directly)

Using `RedshiftOperator` or directly using psycopg2 (or another database adapter) to execute SQL commands involves manually managing connections and cursors. Here's how this differs:

1. Connection Management: You need to manually create and manage connections and cursors. This increases the risk of connection leaks or other issues if not handled correctly.
2. Flexibility: While this approach offers more control over the database interactions, it requires more boilerplate code and careful management of resources.
3. Integration with Airflow: This method is less integrated with Airflow features like connection pooling, logging, and templating. It can lead to more code for handling what Airflow's operators already provide out of the box.
4. Error Handling and Monitoring: You might need to implement additional error handling and logging to ensure that SQL execution issues are captured and visible in the Airflow UI.

When to Choose One Over the Other

- Choose `RedshiftSQLOperator` when you want a straightforward, integrated, and maintainable way to run SQL commands in Redshift within Airflow. It's especially useful if you want to leverage Airflow's built-in capabilities like connection management, templating, and logging.
- Choose `RedshiftOperator` or direct psycopg2 usage if you need very specific control over the database interactions that aren't covered by the `RedshiftSQLOperator`, or when executing complex transactions or session-specific configurations that require manual cursor and connection management.

In summary, for most use cases involving SQL execution in Redshift within an Airflow context, the `RedshiftSQLOperator` is recommended due to its simplicity, robust integration with Airflow's features, and easier maintainability.