

Пайплайны

Оглавление

- > [Оглавление](#)
- > [Создание DAG'a](#)
 - [Вариант 1](#)
 - [Вариант 2](#)
 - [Вариант 3](#)
- > [Аргументы DAG](#)
 - [Trigger Rule](#)
- > [Хуки, операторы, сенсоры](#)
 - [CONNECTIONS](#)
 - [Операторы](#)
 - [Сенсоры](#)
- > [Ветвление](#)
 - [Операторы ветвления](#)
 - [BranchPythonOperator](#)
 - [ShortCircuitOperator](#)
 - [BranchDateTimeOperator](#)
- > [Шаблоны Jinja и макросы](#)
 - [Шаблонизация](#)
 - [Макросы](#)
 - [Пользовательские макросы](#)

Создание DAG'a

Вариант 1

```
dag = DAG("ais_simple_dag_v2",
          schedule_interval='@daily',
          default_args=DEFAULT_ARGS,
          max_active_runs=1,
          tags=['sokolov']
        )
wait_until_6am = TimeDeltaSensor(
    task_id='wait_until_6am',
    delta=timedelta(seconds=6 * 60 * 60),
    dag=dag
)
```

Самый простой вариант. Создание переменной класса [DAG](#). В данном случае каждому таску надо обязательно привязываться к [DAG](#)'у. И внутри каждого таска явно указывать на DAG.

Вариант 2

```
with DAG(
    dag_id='ais_simple_dag',
    schedule_interval='@daily',
    default_args=DEFAULT_ARGS,
    max_active_runs=1,
    tags=['sokolov']
) as dag:

    wait_until_6am = TimeDeltaSensor(
        task_id='wait_until_6am',
        delta=timedelta(seconds=6 * 60 * 60)
    )
```

Он аналогичен первому варианту, но производится через контекстный менеджер. Здесь [DAG](#) не нужно указывать внутри каждого таска, он назначается им автоматически.

Вариант 3

```
@dag(
    start_date=days_ago(12),
    dag_id='sokol_simple_dag',
    schedule_interval='@daily',
    default_args=DEFAULT_ARGS,
    max_active_runs=1,
    tags=['sokolov']
)
def generate_dag():
    wait_until_6am = TimeDeltaSensor(
        task_id='wait_until_6am',
        delta=timedelta(seconds=6 * 60 * 60)
    )
dag = generate_dag()
```

Этот вариант появился в Airflow2. **DAG** создается с помощью декоратора `@dag`. Нам требуется создать функцию со списком задач внутри и обернуть ее в декоратор. Таким образом мы получаем переменную класса **DAG**. Эту переменную мы объявляем в глобальной области видимости, с помощью чего Airflow понимает, что внутри скрипта находится DAG.

Аргументы DAG

```
DEFAULT_ARGS = {
    'owner': 'ais',
    'queue': 'ais_queue',
    'pool': 'user_pool',
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'depends_on_past': False,
    'wait_for_downstream': False,
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
    'priority_weight': 10,
    'start_date': datetime(2021, 1, 1),
    'end_date': datetime(2025, 1, 1),
    'sla': timedelta(hours=2),
    'execution_timeout': timedelta(seconds=300),
    'on_failure_callback': some_function,
    'on_success_callback': some_other_function,
    'on_retry_callback': another_function,
    'sla_miss_callback': yet_another_function,
    'trigger_rule': 'all_success'
}
```

Аргументы для DAG передаются в виде словаря при создании. Он нужен для того, чтобы не только какой-то конкретный DAG знал о своем поведении, но и все задачи унаследовали это поведение, когда создавались под конкретным DAG'ом.

Подробнее об аргументах

- `owner` - отображает владельца DAG'a в интерфейсе.
- `queue` - отвечает за очередь, в которую становится задача. В случае нескольких воркеров, мы можем направить задачи на разные воркеры. Например, если у какого-то воркера есть сетевой доступ к ресурсам, которых нет во втором. Также можно управлять нагрузкой таким образом.
- `pool` - пул, в рамках которого выполняется задача.
- `email` - нужен для оповещения о падении и запуске задачи.
- `email_on_failure` - флаг для оповещения в случае падения
- `email_on_retry` - флаг для оповещения в случае перезапуска
- `depends_on_past` - задача в данном инстансе будет запущена только в тот момент, когда этот же задача в предыдущем DAG инстансе (в инстансе за предыдущий период) уже была отработана. То есть у нас не может быть два одинаковых задачи в разных инстансах запущено одновременно.

DAG

- `wait_for_downstream` - похож на предыдущий аргумент, только здесь мы ждем не окончания работы данного конкретного задачи, а окончания всех задач, которые зависят от этого. Например, мы забираем данные из источника и складываем их во временную таблицу. Этим занимается одна задача, а следующая задача перекладывает данные из временной таблицы в постоянное хранилище. Таким образом, если бы у нас был включен только аргумент `depends_on_past`, то данные загружались бы действительно по очереди, но в тот момент, когда запускался бы второй задача (более свежий) с загрузкой данных из источника, он бы сначала стер все данные за вчерашний день (ту самую таблицу, из которой в этот момент следующая задача забирал бы данные для переноса их в хранилище). Непонятно, кто бы победил в этой гонке, но мы бы могли потенциально получить неполные данные за предыдущий день в хранилище.
- `retries` - количество перезапусков задачи в случае падения (1 запуск всегда основной + переданное количество в аргументе).

- `retry_delay` - время между попытками перезапуска.
- `priority_weight` - вес приоритета этого taska перед другими.
- `start_date` - время первого запуска. Если это не сегодняшний день, то DAG отработает ровно столько раз, сколько прошло времени между указанным `start_date` и текущей датой. Если передать `airflow.utils.dates.days_ago(3)`, это будет значить, что будет запущено только 3 последних инстанса (за 3 последних дня).
- `end_date` - дата, после которой инстансы перестанут генерироваться.
- `sla` - время, до которого мы ожидаем, что task завершится. Если этого не случится, придет оповещение и в интерфейсе SLA появится запись о том, что этот task завершился не вовремя.
- `execution_timeout` - максимальное время выполнения taska. Если task не успеет выполниться за это время, он будет помечен как FAILED.
- `on_failure_callback` - вызов переданной функции в случае падения
- `on_success_callback` - вызов переданной функции в случае успешного завершения
- `on_retry_callback` - вызов переданной функции в случае перезапуска
- `sla_miss_callback` - вызов переданной функции в случае пропущенного SLA
- `trigger_rule` - ...

Trigger Rule

Он отвечает за то, в каком состоянии должны быть предыдущие taskи, чтобы task, который от них зависит, завершился. По умолчанию это значение `all_success`.

Возможные значения аргумента `trigger_rule`

- `all_success` - следующий task запускается только, когда все предыдущие отработали успешно.
- `all_failed` - все предыдущие taskи должны перейти в состояние FAILED, иначе текущий не встанет в очередь.
- `all_done` - все предыдущие taskи должны завершиться с любым результатом.
- `one_failed` - текущий task начнет свою работу в тот момент, когда хотя бы один предыдущий task перейдет состояние FAILED
- `one_success` - текущий task начнет свою работу в тот момент, когда хотя бы один предыдущий task перейдет состояние SUCCESS

- `none_failed` - ни один из задач, от которых зависит текущий не должен быть в состоянии FAILED или UPSTREAM FAILED
- `none_failed_or_skipped` - ни один из задач, от которых зависит текущий не должен быть в состоянии FAILED, UPSTREAM FAILED или SKIPPED
- `none_skipped` - ни один из задач, от которых зависит текущий не должен быть в состоянии SKIPPED
- `dummy` - задачи могут быть в любом состоянии

Пример использования:

```
end = DummyOperator(
    task_id='end',
    trigger_rule='one_success'
)
```

Хуки, операторы, сенсоры

CONNECTIONS

Соединения нужны для системного управления параметрами подключения к различным системам. У каждого `connection` есть уникальный ключ (`conn_id`) или его название.

Их можно использовать из кода напрямую или с помощью хуков.

Пример, в котором мы с помощью `BaseHook` вытаскиваем пароль из конкретного соединения.

```
from airflow.hooks import BaseHook
import logging
logging.info(BaseHook.get_connection('conn_karpov_mysql').password)
```

Хук - это интерфейс для соединения. В нем скрывается low-level код для работы с источником, а вся основная логика обычно передается оператору.

Какие бывают хуки:

- `S3Hook`
- `DockerHook`

- HDFSHook
- HttpHook
- MsSqlHook
- MySqlHook
- OracleHook
- PigCliHook
- PostgresHook
- SqliteHook

Операторы

Операторы - параметризуемые шаблоны для задач. Все, что делается внутри операторов, можно повторить внутри PythonOperator.

Какие бывают операторы:

- BashOperator
- PythonOperator
- EmailOperator
- PostgresOperator
- MySqlOperator
- MsSqlOperator
- HiveOperator
- SimpleHttpOperator
- SlackAPIOperator
- PrestoToMySqlOperator
- TriggerDagRunOperator

Сенсоры

Напомним, что сенсоры ожидают момента наступления какого-либо события.

Параметры сенсора:

- `timeout` - время в секундах, прежде чем сенсор перейдет в состояние FAILED
- `soft_fail` - флаг, при поднятии которого сенсор при неудаче переходит в состояние SKIPPED
- `poke_interval` - время в секундах между попытками, в которые конкретный сенсор будет выяснять, отработало ли событие
- `mode` - `poke` - всегда держит занятым конкретный воркер | `reschedule` - когда сенсор отработал и понял, что событие, которое он ожидает, еще не наступило, он освобождает воркер и в следующий раз занимает его только после того, как прошло время, указанное в `poke_interval`.

В Airflow2 появился экспериментальный способ запуска сенсоров - `smart sensor` - эта настройка включается для всего Airflow и в тот момент, когда она включена, сенсоры начинают запускаться в отдельном одном или нескольких (в зависимости от настроек) процессе по очереди. Мы в едином месте по очереди запускаем сенсоры, чтобы выяснить какое из событий отработало и соответствующий сенсор переводим в состояние SUCCESS в тот момент, когда оно произошло. Таким образом заимствуется всего несколько воркеров для того, чтобы покрыть все DAG'и сенсорами.

Какие бывают сенсоры:

- `ExternalTaskSensor` - логически связывает между собой DAG'и
- `SqlSensor` - дожидается, когда в результатах запроса вернется хотя бы одна строка
- `TimeDeltaSensor` - дожидается, когда пройдет определенное количество секунд после времени запуска DAG'a
- `HdfsSensor` - ждет появления определенного файла в папке внутри HDFS
- `PythonSensor` - ждет, когда python функция вернет значение True
- `DayOfWeekSensor` - ждет наступления конкретного дня

Подробнее про `ExternalTaskSensor`

```
is_payments_done = ExternalTaskSensor(
    task_id='is_payments_done',
    external_dag_id='load_payments',
    external_task_id='end',
    timeout=600,
    allowed_status=['success'],
    failed_states=['failed', 'skipped'],
    mode='reschedule'
)
```

Например, если в одном DAG'e забираются данные из источника и складываются в хранилище, а в другом поверх этих данных строятся какие-то витрины, то оба этих DAG'a должны быть логически связаны. Складывать их в единый DAG - не очень правильно, потому что они относятся к различным уровням хранилища.

[ExternalTaskSensor](#) в данном случае поможет внутри DAG'a с витринами дожидаться момента, когда завершится DAG с получением данных из источника.

Ветвление

Операторы ветвления

- BranchPythonOperator
- ShortCircuitOperator
- BranchDateTimeOperator

BranchPythonOperator

Логика ветвления реализуется в основном с помощью данного оператора.

```
def select_random_func():
    return random.choice(['task_1', 'task_2', 'task_3'])

start = DummyOperator(task_id='start')

select_random = BranchPythonOperator(
    task_id='select_random',
    python_callable=select_random_func
)

task_1 = DummyOperator(task_id='task_1')
task_2 = DummyOperator(task_id='task_2')
task_3 = DummyOperator(task_id='task_3')

start >> select_random >> [task_1, task_2, task_3]
```

Функция, поверх которой работает [BranchPythonOperator](#) должна вернуть название одного или нескольких задач, которые начнут работать после завершения этого задачи. Все задачи, которые не будут упомянуты в выводе из этой функции, перейдут в состояние SKIPPED и будут пропущены. Если же функция вернет None, то будут пропущены все задачи, которые зависят от этого задачи. В данном примере мы случайным образом выбираем один из трех последующих задач.

ShortCircuitOperator

Возвращает значение **True** или **False**. В первом случае все последующие задачи выполняются, в обратном - нет.

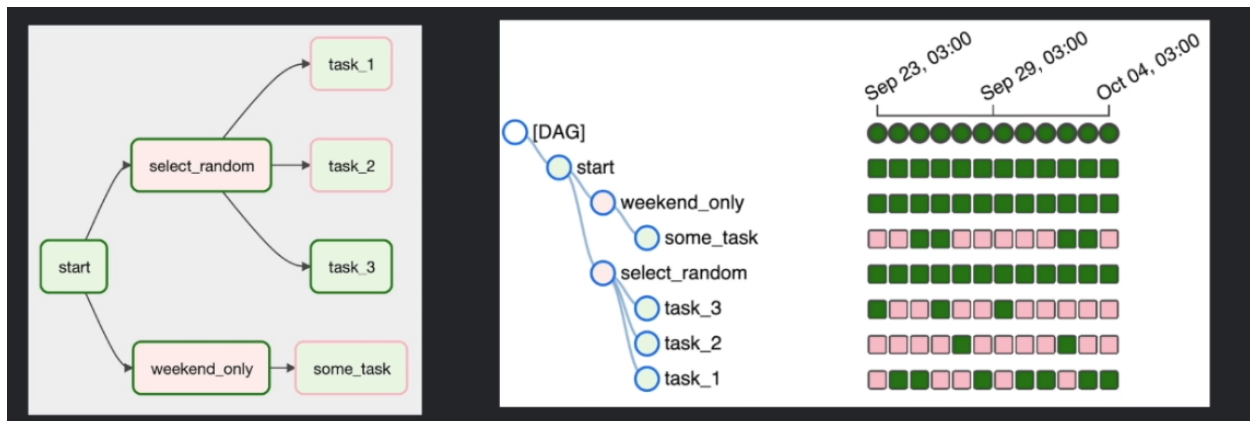
```
def is_weekend_func(execution_dt):
    exec_day = datetime.strptime(execution_dt, '%Y-%m-%d').weekday()
    return exec_day in [5, 6]

weekend_only = ShortCircuitOperator(
    task_id='weekend_only',
    python_callable=is_weekend_func,
    op_kwargs={'execution_dt': {{ ds }}}
)

some_task = DummyOperator(task_id='some_task')

start >> weekend_only >> some_task
```

В примере все последующие задачи запускаются только в выходные (5 и 6 день).



Отображение в виде графа/дерева

- **some_task** чередует 5 пропусков и 2 запуска
- **task_1**, **task_2**, **task_3** отрабатывают случайным образом

BranchDateTimeOperator

```
dummy_task_1 = DummyOperator(task_id='date_in_range', dag=dag)
dummy_task_2 = DummyOperator(task_id='date_outside_range', dag=dag)

cond1 = BranchDateTimeOperator(
    task_id='datetime_branch',
    follow_task_ids_if_true=['date_in_range'],
```

```

follow_task_ids_if_false=['date_outside_range'],
target_upper=datetime.datetime(2020, 10, 10, 15, 0, 0),
target_lower=datetime.datetime(2020, 10, 10, 14, 0, 0),
dag=dag
)

#Run dummy_task_1 if cond1 executes between 2020-10-10 14:00:00 and 2020-10-10 15:00:00
cond1 >> [dummy_task_1, dummy_task_2]

```



Данный оператор запускает тот или иной task в зависимости от того, попадает ли execution date в указанный промежуток времени. Если execution date между `target_lower` и `target_upper`, то запускается task `date_in_range`, иначе `date_outside_range`.

Шаблоны Jinja и макросы

Шаблонизация

В Airflow для параметризации внутри DAG'ов используется [Jinja](#). Параметризуемое выражение окружается двойными фигурными скобками и в момент определения раскрывается с помощью шаблонизатора.

Шаблоны

 Шаблон	 Расшифровка
<code>{{ execution_date }}</code>	execution_date
<code>{{ ds }}</code>	execution_date (YYYY-MM-DD)
<code>{{ ds_nodash }}</code>	execution_date (YYYYMMDD)
<code>{{ ts }}</code>	execution_date (2021-01-01T00:00:00+00:00)
<code>{{ yesterday_ds }}</code>	Вчерашний день относительно execution_date
<code>{{ tomorrow_ds }}</code>	Завтрашний день относительно execution_date
<code>{{ var.value.my_var }}</code>	Значение ключа в глобальной переменной (словарь)
<code>{{ var.json.my_var.path }}</code>	Значение ключа в глобальной переменной (json)
<code>{{ conf }}</code>	airflow.cfg



Посмотреть во что именно раскрылся шаблон можно двумя способами:

- через CLI
- во вкладке Rendered конкретного taska

Макросы

Кроме predefined параметров можно еще использовать внутри шаблонов несколько python-пакетов.

Макросы

 Переменная	 Пакет в Питоне
<u>macros.datetime</u>	datetime.datetime
<u>macros.timedelta</u>	datetime.timedelta
<u>macros.dateutil</u>	dateutil
<u>macros.time</u>	datetime.time
<u>macros.uuid</u>	uuid
<u>macros.random</u>	random

Примеры

```
'{{ macros.datetime.now() }}' # возвращает текущее время из пакета datetime.datetime
'{{ execution_date - macros.timedelta(days=5) }}' # возвращает дату за 5 дней до текущего запуска
'{{ macros.ds_add(ds, -4) }}' # то же самое, только 4 дня
```

Пользовательские макросы

```
def some_custom_func():...

dag = DAG("dina_branches",
          schedule_args='@daily',
          max_active_runs=1,
          tags=['karpov'],
          user_defined_macros={'my_custom_macro': some_custom_func})

bo = BashOperator(
    task_id='my_task',
    bash_command="echo {{ my_custom_macro }}",
    dag=dag
)
```

Для использования пользовательских макросов нужно написать функцию, которая возвращает требуемое значение, прописать эту функцию при создании DAG'a в параметр `user_defined_macros` и дальше использовать эту функцию, как и любой другой template.

Какие поля проходят через шаблонизатор

```
class BashOperator(BaseOperator):
    """Execute a Bash script, command or set of commands...."""

    template_fields = ('bash_command', 'env')
    template_fields_renderers = {'bash_command': 'bash', 'env': 'json'}
    template_ext = (
        '.sh',
        '.bash',
    )
    ui_color = '#f0ede4'

    def __init__(
        self,
        *,
        bash_command: str,
        env: Optional[Dict[str, str]] = None
        output_encoding: str = 'utf-8',
        skip_exit_code: int = 99,
        **kwargs,
    ) -> None:...
```

Данный код описывает `BashOperator`. Применяя его, мы можем видеть, что в `template_fields` перечислены поля, которые проходят через шаблонизатора (`bash_command` и `env`). В `template_fields_renderers` видим, в каком виде в интерфейсе Airflow будет отображаться то или иное поле. В `template_ext` хранится список расширений файлов, в которые мы можем заглянуть и заменить внутри этого файла для исполнения шаблона на конкретное значение.

Примеры

```
template_str = dedent("""
-----
ds: {{ ds }}
ds_nodash: {{ ds_nodash }}
ts: {{ ts }}
gv_karpov: {{ var.value.gv_karpov }}
gv_karpov, course: {{ var.json.gv_karpov_json.course }}

5 дней назад: {{ macros.ds_add(ds, -5) }}
только год: {{ macros.ds_format(ds, "%Y-%m-%d", "%Y") }}
unixtime: {{ "{:.0f}".format(marcos.time.mktime(execution_date.timetuple())*1000) }}
-----
""")

def print_template_func(print_this):
    logging_info(print_this)
```

```
print_templates = PythonOperator(
    task_id='print_templates',
    python_callable=print_template_func,
    op_args=[template_str]
)
```

В данном случае `PythonOperator` выводит в лог значение переменной `template_str`.
Посмотрим, как будут выглядеть шаблоны.

```
*** Reading local file: /var/log/airflow/dina_examples/print_templates/2021-10-08T00:00:00+00:00/8.log
[2021-10-09 22:13:05,566] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: dina_examples.print_templates
[2021-10-09 22:13:05,605] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: dina_examples.print_templates
[2021-10-09 22:13:05,605] {taskinstance.py:1095} INFO -
-----
[2021-10-09 22:13:05,606] {taskinstance.py:1096} INFO - Starting attempt 8 of 8
[2021-10-09 22:13:05,606] {taskinstance.py:1097} INFO -
-----
[2021-10-09 22:13:05,626] {taskinstance.py:1115} INFO - Executing <Task(PythonOperator): print_templates> on 2021-10-08T00:00:00+00:00
[2021-10-09 22:13:05,637] {standard_task_runner.py:52} INFO - Started process 603090 to run task
[2021-10-09 22:13:05,645] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'dina_examples', 'print_templates']
[2021-10-09 22:13:05,648] {standard_task_runner.py:77} INFO - Job 718: Subtask print_templates
[2021-10-09 22:13:05,752] {logging_mixin.py:109} INFO - Running <TaskInstance: dina_examples.print_templates 2021-10-08T00:00:00+00:00>
[2021-10-09 22:13:05,932] {taskinstance.py:1254} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=Karpov
AIRFLOW_CTX_DAG_ID=dina_examples
AIRFLOW_CTX_TASK_ID=print_templates
AIRFLOW_CTX_EXECUTION_DATE=2021-10-08T00:00:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-10-08T00:00:00+00:00
[2021-10-09 22:13:05,935] {dina_examples.py:71} INFO -
-----
ds: 2021-10-08
ds_nodash: 20211008
ts: 2021-10-08T00:00:00+00:00
gv_karpov: {'one': 1, 'two': 2}
gv_karpov, course: ETL

5 дней назад: 2021-10-03
только год: 2021
unixtime: 1633651200000
-----
[2021-10-09 22:13:05,935] {python.py:151} INFO - Done. Returned value was: None
[2021-10-09 22:13:05,955] {taskinstance.py:1219} INFO - Marking task as SUCCESS. dag_id=dina_examples, task_id=print_templates
[2021-10-09 22:13:06,023] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-10-09 22:13:06,072] {local_task_job.py:261} INFO - 0 downstream tasks scheduled from follow-on schedule check
```

Все, окруженное в двойные фигурные скобки, раскрылось в конкретное значение.

Передача аргументов

Аргументы для PythonOperator

Их можно передавать следующими способами:

- `op_args`
- `op_kwargs`

- `template_dict`
- `provide_context`

Рассмотрим на примере

```
def print_args_func(arg1, arg2, **kwargs):
    logging.info('-----')
    logging.info(f'op_args, №1: {arg1}')
    logging.info(f'op_args, №2: {arg2}')
    logging.info(f'op_kwargs, №1: ' + kwargs['kwarg1'])
    logging.info(f'op_kwargs, №2: ' + kwargs['kwarg2'])
    logging.info(f'template_dict, gv_karpov: ' + kwargs['templates_dict']['gv_karpov'])
    logging.info(f'templates_dict, task_owner: ' + kwargs['templates_dict']['task_owner'])
    logging.info(f'context, {{ ds }}: ' + kwargs['ds'])
    logging.info(f'context, {{ tomorrow_ds }}: ' + kwargs['tomorrow_ds'])
    logging.info('-----')

print_args = PythonOperator(
    task_id='print_args',
    python_callable=print_args_func,
    op_args=['arg1', 'arg2'],
    op_kwargs={'kwarg1': 'kwarg1', 'kwarg2': 'kwarg2'},
    templates_dict={'gv_karpov': '{{ var.value.gv_karpov }}',
                    'task_owner': '{{ task.owner }}'},
    provide_context=True
)
```

В данной функции передается 2 позиционных аргумента `arg1` и `arg2`, а также словарь с ключами `kwarg1` и `kwarg2`. Также передаем еще один словарь `templates_dict`.

Посмотрим на результат.

```
*** Reading local file: /var/log/airflow/dina_examples/print_args/2021-09-27T00:00:00+00:00/7.log
[2021-10-09 20:32:20,178] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: dina_examples.print_args 2021-09-27T00:00:00+00:00 [queued]>
[2021-10-09 20:32:20,195] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: dina_examples.print_args 2021-09-27T00:00:00+00:00 [queued]>
[2021-10-09 20:32:20,195] {taskinstance.py:1095} INFO -

[2021-10-09 20:32:20,195] {taskinstance.py:1096} INFO - Starting attempt 7 of 7
[2021-10-09 20:32:20,195] {taskinstance.py:1097} INFO -

[2021-10-09 20:32:20,222] {taskinstance.py:1115} INFO - Executing <Task(PythonOperator): print_args> on 2021-09-27T00:00:00+00:00
[2021-10-09 20:32:20,230] {standard_task_runner.py:52} INFO - Started process 561941 to run task
[2021-10-09 20:32:20,240] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'dina_examples', 'print_args', '2021-09-27T00:00:00+00:00', '--job-id=661']
[2021-10-09 20:32:20,243] {standard_task_runner.py:77} INFO - Job 661: Subtask print_args
[2021-10-09 20:32:20,350] {logging_mixin.py:109} INFO - Running <TaskInstance: dina_examples.print_args 2021-09-27T00:00:00+00:00 [running]> on host b3ed1b130bd9
[2021-10-09 20:32:20,509] {taskinstance.py:1254} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=Karpov
AIRFLOW_CTX_DAG_ID=dina_examples
AIRFLOW_CTX_TASK_ID=print_args
AIRFLOW_CTX_EXECUTION_DATE=2021-09-27T00:00:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled_2021-09-27T00:00:00+00:00
[2021-10-09 20:32:20,510] {dina_examples.py:32} INFO - -----
[2021-10-09 20:32:20,511] {dina_examples.py:33} INFO - op_args, №1: arg1
[2021-10-09 20:32:20,511] {dina_examples.py:34} INFO - op_args, №2: arg2
[2021-10-09 20:32:20,511] {dina_examples.py:35} INFO - op_kwargs, №1: kwarg1
[2021-10-09 20:32:20,511] {dina_examples.py:36} INFO - op_kwargs, №2: kwarg2
[2021-10-09 20:32:20,511] {dina_examples.py:37} INFO - templates_dict, gv_karpov: {'one': 1, 'two': 2}
[2021-10-09 20:32:20,511] {dina_examples.py:38} INFO - templates_dict, task.owner: Karpov
[2021-10-09 20:32:20,511] {dina_examples.py:39} INFO - context, {{ ds }}: 2021-09-27
[2021-10-09 20:32:20,511] {dina_examples.py:40} INFO - context, {{ tomorrow_ds }}: 2021-09-28
[2021-10-09 20:32:20,511] {dina_examples.py:41} INFO - -----
[2021-10-09 20:32:20,511] {python.py:151} INFO - Done. Returned value was: None
[2021-10-09 20:32:20,534] {taskinstance.py:1219} INFO - Marking task as SUCCESS. dag_id=dina_examples, task_id=print_args, execution_date=20210927T000000, start_date=2021-10-09 20:32:20,614} {local_task_job.py:151} INFO - Task exited with return code 0
[2021-10-09 20:32:20,666] {local_task_job.py:261} INFO - 0 downstream tasks scheduled from follow-on schedule check
```

