Macros reference

Variables and macros can be used in templates (see the Jinja Templating section)

The following come for free out of the box with Airflow. Additional custom macros can be added globally through Plugins, or at a DAG level through the `DAG.user_defined_macros` argument.

Default Variables

The Airflow engine passes a few variables by default that are accessible in all templates

Variable	Description
`{{ ds }}`	the execution date as `YYYY-MM-DD`
`{{ ds_nodash }}`	the execution date as `YYYYMMDD`
`{{ prev_ds }}`	the previous execution date as `YYYY-MM-DD` if `{{ ds }}` is `2018-01-08` and `schedule_interval` is `@weekly` , `{{ prev_ds }}` will be `2018-01-01`
`{{ prev_ds_nodash }}`	the previous execution date as `YYYYMMDD` if exists, else `None`
`{{ next_ds }}`	the next execution date as `YYYY-MM-DD` if `{{ ds }}` is `2018-01-01` and `schedule_interval` is `@weekly` , `{{ next_ds }}` will be `2018-01-08`
`{{ next_ds_nodash }}`	the next execution date as `YYYYMMDD` if exists, else `None`
`{{ yesterday_ds }}`	the day before the execution date as `YYYY-MM-DD`
`{{ yesterday_ds_nodash }}`	the day before the execution date as `YYYYMMDD`
`{{ tomorrow_ds }}`	the day after the execution date as `YYYY-MM-DD`
`{{ tomorrow_ds_nodash }}`	the day after the execution date as `YYYYMMDD`
`{{ ts }}`	same as `execution_date.isoformat()` . Example: `2018-01-01T00:00:00+00:00`
`{{ ts_nodash }}`	same as `ts` without `-` , `:` and TimeZone info. Example: `20180101T000000`
`{{ ts_nodash_with_tz }}`	same as `ts` without `-` and `:` . Example: `20180101T000000+0000`
`{{ execution_date }}`	the execution_date (logical date) (pendulum.Pendulum)
`{{ prev_execution_date }}`	the previous execution date (if available) (pendulum.Pendulum)
`{{ prev_execution_date_success }}`	execution date from prior successful dag run (if available) (pendulum.Pendulum)
`{{ prev_start_date_success }}`	start date from prior successful dag run (if available) (pendulum.Pendulum)
`{{ next_execution_date }}`	the next execution date (pendulum.Pendulum)
`{{ dag }}`	the DAG object
`{{ task }}`	the Task object
`{{ macros }}`	a reference to the macros package, described below
`{{ task_instance }}`	the task_instance object
`{{ ti }}`	same as `{{ task_instance }}`

Variable	Description
`{{ params }}`	a reference to the user-defined params dictionary which can be overridden by the dictionary passed through `trigger_dag -c` if you enabled `dag_run_conf_overrides_params` in ``airflow.cfg`
`{{ var.value.my_var }}`	global defined variables represented as a dictionary
`{{ var.json.my_var.path }}`	global defined variables represented as a dictionary with deserialized JSON object, append the path to the key within the JSON object
`{{ task_instance_key_str }}`	a unique, human-readable key to the task instance formatted `{dag_id}__{task_id}__{ds_nodash}`
`{{ conf }}`	the full configuration object located at `airflow.configuration.conf` which represents the content of your `airflow.cfg`
`{{ run_id }}`	the `run_id` of the current DAG run
`{{ dag_run }}`	a reference to the DagRun object
`{{ test_mode }}`	whether the task instance was called using the CLI's test subcommand

Note that you can access the object's attributes and methods with simple dot notation. Here are some examples of what is possible: `{{ task.owner }}`, `{{ task.task_id }}`, `{{ ti.hostname }}`, … Refer to the models documentation for more information on the objects' attributes and methods.

The `var` template variable allows you to access variables defined in Airflow's UI. You can access them as either plain-text or JSON. If you use JSON, you are also able to walk nested structures, such as dictionaries like: `{{ var.json.my_dict_var.key1 }}`.

It is also possible to fetch a variable by string if needed with `{{ var.value.get('my.var', 'fallback') }}` or `{{ var.json.get('my.dict.var', {'key1': 'val1'}) }}`. Defaults can be supplied in case the variable does not exist.

Macros

Macros are a way to expose objects to your templates and live under the `macros` namespace in your templates.

A few commonly used libraries and methods are made available.

Variable	Description
`macros.datetime`	The standard lib's **`datetime.datetime`**
`macros.timedelta`	The standard lib's **`datetime.timedelta`**
`macros.dateutil`	A reference to the `dateutil` package
`macros.time`	The standard lib's **`datetime.time`**
`macros.uuid`	The standard lib's **`uuid`**
`macros.random`	The standard lib's **`random`**

Some airflow specific macros are also defined:

Macros.

`airflow.macros.datetime_diff_for_humans(dt, since=None)[source]`

Return a human-readable/approximate difference between two datetimes, or one and now.

Parameters

- **dt** (*datetime.datetime*) – The datetime to display the diff for
- **since** (*None* or *datetime.datetime*) – When to display the date from. If **`None`** then the diff is between **`dt`** and now.

Return type

str

`airflow.macros.ds_add(ds, days)[source]`

Add or subtract days from a YYYY-MM-DD

Parameters

- **ds** (*str*) – anchor date in `YYYY-MM-DD` format to add to
- **days** (*int*) – number of days to add to the ds, you can use negative values

```
>>> ds_add('2015-01-01', 5)
'2015-01-06'
>>> ds_add('2015-01-06', -5)
'2015-01-01'
```

`airflow.macros.ds_format(ds, input_format, output_format)[source]`

Takes an input string and outputs another string as specified in the output format

Parameters

- **ds** (*str*) – input string which contains a date
- **input_format** (*str*) – input string format. E.g. %Y-%m-%d
- **output_format** (*str*) – output string format E.g. %Y-%m-%d

```
>>> ds_format('2015-01-01', "%Y-%m-%d", "%m-%d-%y")
'01-01-15'
>>> ds_format('1/5/2015', "%m/%d/%Y",  "%Y-%m-%d")
'2015-01-05'
```

`airflow.macros.random() → x in the interval [0, 1).`

`airflow.macros.hive.closest_ds_partition(table, ds, before=True, schema='default', metastore_conn_id='metastore_default')[source]`

This function finds the date in a list closest to the target date. An optional parameter can be given to get the closest before or after.

Parameters

- **table** (*str*) – A hive table name
- **ds** (*list[datetime.date]*) – A datestamp `%Y-%m-%d` e.g. `yyyy-mm-dd`
- **before** (*bool or None*) – closest before (True), after (False) or either side of ds
- **schema** (*str*) – table schema
- **metastore_conn_id** (*str*) – which metastore connection to use

Returns

The closest date

Return type

str or None

```
>>> tbl = 'airflow.static_babynames_partitioned'
>>> closest_ds_partition(tbl, '2015-01-02')
'2015-01-01'
```

```
airflow.macros.hive.max_partition(table, schema='default', field=None, filter_map=None, metastore_conn_id='metastore_default')[source]
```

Gets the max partition for a table.

Parameters

- **schema** (*str*) – The hive schema the table lives in
- **table** (*str*) – The hive table you are interested in, supports the dot notation as in “my_database.my_table”, if a dot is found, the schema param is disregarded
- **metastore_conn_id** (*str*) – The hive connection you are interested in. If your default is set you don’t need to use this parameter.
- **filter_map** (*dict*) – partition_key:partition_value map used for partition filtering, e.g. {‘key1’: ‘value1’, ‘key2’: ‘value2’}. Only partitions matching all partition_key:partition_value pairs will be considered as candidates of max partition.
- **field** (*str*) – the field to get the max value from. If there’s only one partition field, this will be inferred

```
>>> max_partition('airflow.static_babynames_partitioned')
'2015-01-01'
```

Previous Next

Was this entry helpful?

☆ ☆ ☆ ☆ ☆

Want to be a part of Apache Airflow? Join community