airflow.models.baseoperator

Base operator for all operators.

Module Contents

airflow.models.baseoperator.ScheduleInterval[source]

airflow.models.baseoperator.TaskStateChangeCallback[source]

class **airflow.models.baseoperator.BaseOperatorMeta**[source]

Bases: abc.ABCMeta

Base metaclass of BaseOperator.

__call__(*cls*, **args*, ***kwargs*)[source]

Called when you call BaseOperator(). In this way we are able to perform an action after initializing an operator no matter where the `super().__init__` is called (before or after assign of new attributes in a custom operator).

class **airflow.models.baseoperator.BaseOperator**(*task_id: str*, *owner: str = conf.get('operators', 'DEFAULT_OWNER')*, *email: Optional[Union[str, Iterable[str]]] = None*, *email_on_retry: bool = conf.getboolean('email', 'default_email_on_retry', fallback=True)*, *email_on_failure: bool = conf.getboolean('email', 'default_email_on_failure', fallback=True)*, *retries: Optional[int] = conf.getint('core', 'default_task_retries', fallback=0)*, *retry_delay: timedelta = timedelta(seconds=300)*, *retry_exponential_backoff: bool = False*, *max_retry_delay: Optional[datetime] = None*, *start_date: Optional[datetime] = None*, *end_date: Optional[datetime] = None*, *depends_on_past: bool = False*, *wait_for_downstream: bool = False*, *dag=None*, *params: Optional[Dict] = None*,

*default_args: Optional[Dict] = None, priority_weight: int = 1, weight_rule: str = WeightRule.DOWNSTREAM, queue: str = conf.get('celery', 'default_queue'), pool: Optional[str] = None, pool_slots: int = 1, sla: Optional[timedelta] = None, execution_timeout: Optional[timedelta] = None, on_execute_callback: Optional[TaskStateChangeCallback] = None, on_failure_callback: Optional[TaskStateChangeCallback] = None, on_success_callback: Optional[TaskStateChangeCallback] = None, on_retry_callback: Optional[TaskStateChangeCallback] = None, trigger_rule: str = TriggerRule.ALL_SUCCESS, resources: Optional[Dict] = None, run_as_user: Optional[str] = None, task_concurrency: Optional[int] = None, executor_config: Optional[Dict] = None, do_xcom_push: bool = True, inlets: Optional[Any] = None, outlets: Optional[Any] = None, task_group: Optional['TaskGroup'] = None, **kwargs*)[source]

Bases: airflow.models.base.Operator , airflow.utils.log.logging_mixin.LoggingMixin , airflow.models.taskmixin.TaskMixin

Abstract base class for all operators. Since operators create objects that become nodes in the dag, BaseOperator contains many recursive methods for dag crawling behavior. To derive this class, you are expected to override the constructor as well as the 'execute' method.

Operators derived from this class should perform or trigger certain tasks synchronously (wait for completion). Example of operators could be an operator that runs a Pig job (PigOperator), a sensor operator that waits for a partition to land in Hive (HiveSensorOperator), or one that moves data from Hive to MySQL (Hive2MySqlOperator). Instances of these operators (tasks) target specific operations, running specific scripts, functions or data transfers.

This class is abstract and shouldn't be instantiated. Instantiating a class derived from this one results in the creation of a task object, which ultimately becomes a node in DAG objects. Task dependencies should be set by using the set_upstream and/or set_downstream methods.

Parameters

- **task_id** (*str*) – a unique, meaningful id for the task
- **owner** (*str*) – the owner of the task, using the unix username is recommended
- **email** (*str* or *list*[*str*]) – the 'to' email address(es) used in email alerts. This can be a single email or multiple ones. Multiple addresses can be specified as a comma or semi-colon separated string or by passing a list of strings.
- **email_on_retry** (*bool*) – Indicates whether email alerts should be sent when a task is retried

- **email_on_failure** (*bool*) – Indicates whether email alerts should be sent when a task failed
- **retries** (*int*) – the number of retries that should be performed before failing the task
- **retry_delay** (*datetime.timedelta*) – delay between retries
- **retry_exponential_backoff** (*bool*) – allow progressive longer waits between retries by using exponential backoff algorithm on retry delay (delay will be converted into seconds)
- **max_retry_delay** (*datetime.timedelta*) – maximum delay interval between retries
- **start_date** (*datetime.datetime*) – The `start_date` for the task, determines the `execution_date` for the first task instance. The best practice is to have the start_date rounded to your DAG's `schedule_interval`. Daily jobs have their start_date some day at 00:00:00, hourly jobs have their start_date at 00:00 of a specific hour. Note that Airflow simply looks at the latest `execution_date` and adds the `schedule_interval` to determine the next `execution_date`. It is also very important to note that different tasks' dependencies need to line up in time. If task A depends on task B and their start_date are offset in a way that their execution_date don't line up, A's dependencies will never be met. If you are looking to delay a task, for example running a daily task at 2AM, look into the `TimeSensor` and `TimeDeltaSensor`. We advise against using dynamic `start_date` and recommend using fixed ones. Read the FAQ entry about start_date for more information.
- **end_date** (*datetime.datetime*) – if specified, the scheduler won't go beyond this date
- **depends_on_past** (*bool*) – when set to true, task instances will run sequentially and only if the previous instance has succeeded or has been skipped. The task instance for the start_date is allowed to run.
- **wait_for_downstream** (*bool*) – when set to true, an instance of task X will wait for tasks immediately downstream of the previous instance of task X to finish successfully or be skipped before it runs. This is useful if the different instances of a task X alter the same asset, and this asset is used by tasks downstream of task X. Note that depends_on_past is forced to True wherever wait_for_downstream is used. Also note that only tasks *immediately* downstream of the previous task instance are waited for; the statuses of any tasks further downstream are ignored.
- **dag** (*airflow.models.DAG*) – a reference to the dag the task is attached to (if any)
- **priority_weight** (*int*) – priority weight of this task against other task. This allows the executor to trigger higher priority tasks before others when things get backed up. Set priority_weight as a higher number for more important tasks.

- **weight_rule** (*str*) – weighting method used for the effective total priority weight of the task. Options are: `{ downstream | upstream | absolute }` default is `downstream` When set to `downstream` the effective weight of the task is the aggregate sum of all downstream descendants. As a result, upstream tasks will have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and desire to have all upstream tasks to complete for all runs before each dag can continue processing downstream tasks. When set to `upstream` the effective weight is the aggregate sum of all upstream ancestors. This is the opposite where downstream tasks have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and prefer to have each dag complete before starting upstream tasks of other dags. When set to `absolute`, the effective weight is the exact `priority_weight` specified without additional weighting. You may want to do this when you know exactly what priority weight each task should have. Additionally, when set to `absolute`, there is bonus effect of significantly speeding up the task creation process as for very large DAGS. Options can be set as string or using the constants defined in the static class `airflow.utils.WeightRule`
- **queue** (*str*) – which queue to target when running this job. Not all executors implement queue management, the CeleryExecutor does support targeting specific queues.
- **pool** (*str*) – the slot pool this task should run in, slot pools are a way to limit concurrency for certain tasks
- **pool_slots** (*int*) – the number of pool slots this task should use (>= 1) Values less than 1 are not allowed.
- **sla** (*datetime.timedelta*) – time by which the job is expected to succeed. Note that this represents the `timedelta` after the period is closed. For example if you set an SLA of 1 hour, the scheduler would send an email soon after 1:00AM on the `2016-01-02` if the `2016-01-01` instance has not succeeded yet. The scheduler pays special attention for jobs with an SLA and sends alert emails for sla misses. SLA misses are also recorded in the database for future reference. All tasks that share the same SLA time get bundled in a single email, sent soon after that time. SLA notification are sent once and only once for each task instance.
- **execution_timeout** (*datetime.timedelta*) – max time allowed for the execution of this task instance, if it goes beyond it will raise and fail.
- **on_failure_callback** (*TaskStateChangeCallback*) – a function to be called when a task instance of this task fails. a context dictionary is passed as a single parameter to this function. Context contains references to related objects to the task instance and is documented under the macros section of the API.

- **on_execute_callback** (*TaskStateChangeCallback*) – much like the `on_failure_callback` except that it is executed right before the task is executed.
- **on_retry_callback** (*TaskStateChangeCallback*) – much like the `on_failure_callback` except that it is executed when retries occur.
- **on_success_callback** (*TaskStateChangeCallback*) – much like the `on_failure_callback` except that it is executed when the task succeeds.
- **trigger_rule** (*str*) – defines the rule by which dependencies are applied for the task to get triggered. Options are: `{ all_success | all_failed | all_done | one_success | one_failed | none_failed | none_failed_or_skipped | none_skipped | dummy}` default is `all_success` . Options can be set as string or using the constants defined in the static class `airflow.utils.TriggerRule`
- **resources** (*dict*) – A map of resource parameter names (the argument names of the Resources constructor) to their values.
- **run_as_user** (*str*) – unix username to impersonate while running the task
- **task_concurrency** (*int*) – When set, a task will be able to limit the concurrent runs across execution_dates
- **executor_config** (*dict*) –

  Additional task-level configuration parameters that are interpreted by a specific executor. Parameters are namespaced by the name of executor.

  Example: to run this task in a specific docker container through the KubernetesExecutor

```
MyOperator(...,
    executor_config={
        "KubernetesExecutor":
            {"image": "myCustomDockerImage"}
    }
)
```

- **do_xcom_push** (*bool*) – if True, an XCom is pushed containing the Operator's result

template_fields *:Iterable[str] = []* [source]

template_ext *:Iterable[str] = []* [source]

template_fields_renderers *:Dict[str, str]* [source]

ui_color *:str = #fff* [source]

ui_fgcolor *:str = #000* [source]

pool *:str =* [source]

_base_operator_shallow_copy_attrs
:Tuple[str, ...] = ['user_defined_macros', 'user_defined_filters', 'params', '_log']
[source]

shallow_copy_attrs *:Tuple[str, ...] = []* [source]

operator_extra_links *:Iterable['BaseOperatorLink'] = []*
[source]

__serialized_fields *:Optional[FrozenSet[str]]* [source]

_comps[source]

supports_lineage *= False* [source]

__instantiated *= False* [source]

_lock_for_execution *= False* [source]

dag[source]

Returns the Operator's DAG if set, otherwise raises an error

dag_id[source]

Returns dag id if it has one or an adhoc + owner

deps *:Iterable[BaseTIDep]* [source]

Returns the set of dependencies for the operator. These differ from execution context dependencies in that they are specific to tasks and can be extended/overridden by subclasses.

priority_weight_total[source]

Total priority weight for the task. It might include all upstream or downstream tasks. depending on the weight rule.

- WeightRule.ABSOLUTE - only own weight
- WeightRule.DOWNSTREAM - adds priority weight of all downstream tasks
- WeightRule.UPSTREAM - adds priority weight of all upstream tasks

upstream_list[source]

@property: list of tasks directly upstream

upstream_task_ids[source]

@property: set of ids of tasks directly upstream

downstream_list[source]

@property: list of tasks directly downstream

downstream_task_ids[source]

@property: set of ids of tasks directly downstream

task_type[source]

@property: type of the task

roots[source]

Required by TaskMixin

leaves[source]

Required by TaskMixin

output[source]

Returns reference to XCom pushed by current operator

inherits_from_dummy_operator[source]

Used to determine if an Operator is inherited from DummyOperator

__eq__(*self*, *other*)[source]

__ne__(*self*, *other*)[source]

__hash__(*self*)[source]

__or__(*self*, *other*)[source]

Called for [This Operator] | [Operator], The inlets of other will be set to pickup the outlets from this operator. Other will be set as a downstream task of this operator.

__gt__(*self*, *other*)[source]

Called for [Operator] > [Outlet], so that if other is an attr annotated object it is set as an outlet of this Operator.

__lt__(*self*, *other*)[source]

Called for [Inlet] > [Operator] or [Operator] < [Inlet], so that if other is an attr annotated object it is set as an inlet to this operator

__setattr__(*self*, *key*, *value*)[source]

add_inlets(*self*, *inlets: Iterable[Any]*)[source]

Sets inlets to this operator

add_outlets(*self*, *outlets: Iterable[Any]*)[source]

Defines the outlets of this operator

get_inlet_defs(*self*)[source]

Returns

list of inlets defined for this operator

get_outlet_defs(*self*)[source]

Returns

list of outlets defined for this operator

has_dag(*self*)[source]

Returns True if the Operator has been assigned to a DAG.

prepare_for_execution(*self*)[source]

Lock task for execution to disable custom action in __setattr__ and returns a copy of the task

set_xcomargs_dependencies(*self*)[source]

Resolves upstream dependencies of a task. In this way passing an `XComArg` as value for a template field will result in creating upstream relation between two tasks.

Example:

```
with DAG(...):
    generate_content =
GenerateContentOperator(task_id="generate_content")
    send_email = EmailOperator(...,
html_content=generate_content.output)

# This is equivalent to
with DAG(...):
    generate_content =
GenerateContentOperator(task_id="generate_content")
    send_email = EmailOperator(
        ..., html_content="{{
task_instance.xcom_pull('generate_content') }}"
    )
    generate_content >> send_email
```

operator_extra_link_dict(*self*)[source]

Returns dictionary of all extra links for the operator

global_operator_extra_link_dict(*self*)[source]

Returns dictionary of all global extra links

pre_execute(*self*, *context: Any*)[source]

This hook is triggered right before self.execute() is called.

execute(*self*, *context: Any*)[source]

This is the main method to derive when creating an operator. Context is the same dictionary used as when rendering jinja templates.

Refer to get_template_context for more context.

post_execute(*self*, *context: Any*, *result: Any = None*) [source]

This hook is triggered right after self.execute() is called. It is passed the execution context and any results returned by the operator.

on_kill(*self*)[source]

Override this method to cleanup subprocesses when a task instance gets killed. Any use of the threading, subprocess or multiprocessing module within an operator needs to be cleaned up or it will leave ghost processes behind.

__deepcopy__(*self*, *memo*)[source]

Hack sorting double chained task lists by task_id to avoid hitting max_depth on deepcopy operations.

__getstate__(*self*)[source]

__setstate__(*self*, *state*)[source]

render_template_fields(*self*, *context: Dict*, *jinja_env: Optional[jinja2.Environment] = None*)[source]

Template all attributes listed in template_fields. Note this operation is irreversible.

Parameters

- **context** (*dict*) – Dict with values to apply on content
- **jinja_env** (*jinja2.Environment*) – Jinja environment

_do_render_template_fields(*self*, *parent: Any*, *template_fields: Iterable[str]*, *context: Dict*, *jinja_env: jinja2.Environment*, *seen_oids: Set*) [source]

render_template(*self*, *content: Any*, *context: Dict*, *jinja_env: Optional[jinja2.Environment] = None*, *seen_oids: Optional[Set] = None*)[source]

Render a templated string. The content can be a collection holding multiple templated strings and will be templated recursively.

Parameters

- **content** (*Any*) – Content to template. Only strings can be templated (may be inside collection).

- **context** (*dict*) – Dict with values to apply on templated content
- **jinja_env** (*jinja2.Environment*) – Jinja environment. Can be provided to avoid re-creating Jinja environments during recursion.
- **seen_oids** (*set*) – template fields already rendered (to avoid RecursionError on circular dependencies)

Returns

Templated content

_render_nested_template_fields(*self*, *content: Any*, *context: Dict*, *jinja_env: jinja2.Environment*, *seen_oids: Set*)[source]

get_template_env(*self*)[source]

Fetch a Jinja template environment from the DAG or instantiate empty environment if no DAG.

prepare_template(*self*)[source]

Hook that is triggered after the templated fields get replaced by their content. If you need your operator to alter the content of the file before the template is rendered, it should override this method to do so.

resolve_template_files(*self*)[source]

Getting the content of files for template_field / template_ext

clear(*self*, *start_date: Optional[datetime] = None*, *end_date: Optional[datetime] = None*, *upstream: bool = False*, *downstream: bool = False*, *session: Session = None*)[source]

Clears the state of task instances associated with the task, following the parameters specified.

get_task_instances(*self*, *start_date: Optional[datetime] = None*, *end_date: Optional[datetime] = None*, *session: Session = None*)[source]

Get a set of task instance related to this task for a specific date range.

get_flat_relative_ids(*self*, *upstream: bool = False*, *found_descendants: Optional[Set[str]] = None*)[source]

Get a flat set of relatives' ids, either upstream or downstream.

get_flat_relatives(*self*, *upstream: bool = False*)
[source]

Get a flat list of relatives, either upstream or downstream.

run(*self*, *start_date: Optional[datetime] = None*, *end_date: Optional[datetime] = None*, *ignore_first_depends_on_past: bool = True*, *ignore_ti_state: bool = False*, *mark_success: bool = False*)[source]

Run a set of task instances for a date range.

dry_run(*self*)[source]

Performs dry run for the operator - just render template fields.

get_direct_relative_ids(*self*, *upstream: bool = False*)
[source]

Get set of the direct relative ids to the current task, upstream or downstream.

get_direct_relatives(*self*, *upstream: bool = False*)
[source]

Get list of the direct relatives to the current task, upstream or downstream.

__repr__(*self*)[source]

add_only_new(*self*, *item_set: Set[str]*, *item: str*)
[source]

Adds only new items to item set

_set_relatives(*self*, *task_or_task_list: Union[TaskMixin, Sequence[TaskMixin]]*, *upstream: bool = False*)[source]

Sets relatives for the task or task list.

set_downstream(*self*, *task_or_task_list: Union[TaskMixin, Sequence[TaskMixin]]*)[source]

Set a task or a task list to be directly downstream from the current task. Required by TaskMixin.

set_upstream(*self*, *task_or_task_list: Union[TaskMixin, Sequence[TaskMixin]]*)[source]

Set a task or a task list to be directly upstream from the current task. Required by TaskMixin.

static **xcom_push**(*context: Any*, *key: str*, *value: Any*, *execution_date: Optional[datetime] = None*)[source]

Make an XCom available for tasks to pull.

Parameters

- **context** – Execution Context Dictionary
- **key** (*str*) – A key for the XCom
- **value** (*any pickleable object*) – A value for the XCom. The value is pickled and stored in the database.
- **execution_date** (*datetime*) – if provided, the XCom will not be visible until this date. This can be used, for example, to send a message to a task on a future date without it being immediately visible.

Type

Any

static **xcom_pull**(*context: Any*, *task_ids: Optional[List[str]] = None*, *dag_id: Optional[str] = None*, *key: str = XCOM_RETURN_KEY*, *include_prior_dates: Optional[bool] = None*)[source]

Pull XComs that optionally meet certain criteria.

The default value for *key* limits the search to XComs that were returned by other tasks (as opposed to those that were pushed manually). To remove this filter, pass key=None (or any desired value).

If a single task_id string is provided, the result is the value of the most recent matching XCom from that task_id. If multiple task_ids are provided, a tuple of matching values is returned. None is returned whenever no matches are found.

Parameters

- **context** – Execution Context Dictionary
- **key** (*str*) – A key for the XCom. If provided, only XComs with matching keys will be returned. The default key is 'return_value', also available as a constant XCOM_RETURN_KEY. This key is automatically given to XComs returned by tasks (as opposed to being pushed manually). To remove the filter, pass key=None.
- **task_ids** (*str or iterable of strings (representing task_ids)*) – Only XComs from tasks with matching ids will be pulled. Can pass None to remove the filter.

- **dag_id** (*str*) – If provided, only pulls XComs from this DAG. If None (default), the DAG of the calling task is used.
- **include_prior_dates** (*bool*) – If False, only XComs from the current execution_date are returned. If True, XComs from previous dates are returned as well.

Type

Any

extra_links(*self*)[source]

@property: extra links for the task

get_extra_links(*self*, *dttm: datetime*, *link_name: str*)[source]

For an operator, gets the URL that the external links specified in *extra_links* should point to.

Raises

ValueError – The error message of a ValueError will be passed on through to the fronted to show up as a tooltip on the disabled link

Parameters

- **dttm** – The datetime parsed execution date for the URL being searched for
- **link_name** – The name of the link we're looking for the URL for. Should be one of the options specified in *extra_links*

Returns

A URL

classmethod **get_serialized_fields**(*cls*)[source]

Stringified DAGs and operators contain exactly these fields.

is_smart_sensor_compatible(*self*)[source]

Return if this operator can use smart service. Default False.

---

airflow.models.baseoperator.chain(**tasks*)[source]

Given a number of tasks, builds a dependency chain.

Support mix airflow.models.BaseOperator and List[airflow.models.BaseOperator].

If you want to chain between two List[airflow.models.BaseOperator], have to make sure they have same length.

```
chain(t1, [t2, t3], [t4, t5], t6)
```

is equivalent to:

```
  / -> t2 -> t4 \
t1               -> t6
  \ -> t3 -> t5 /
```

```
t1.set_downstream(t2)
t1.set_downstream(t3)
t2.set_downstream(t4)
t3.set_downstream(t5)
t4.set_downstream(t6)
t5.set_downstream(t6)
```

Parameters

tasks (*List[airflow.models.BaseOperator] or airflow.models.BaseOperator*) – List of tasks or List[airflow.models.BaseOperator] to set dependencies

airflow.models.baseoperator.cross_downstream(*from_tasks: Sequence[BaseOperator], to_tasks: Union[BaseOperator, Sequence[BaseOperator]]*)[source]

Set downstream dependencies for all tasks in from_tasks to all tasks in to_tasks.

```
cross_downstream(from_tasks=[t1, t2, t3], to_tasks=[t4, t5,
t6])
```

is equivalent to:

```
t1 ---> t4
   \ /
t2 -X -> t5
   / \
t3 ---> t6
```

```
t1.set_downstream(t4)
t1.set_downstream(t5)
t1.set_downstream(t6)
t2.set_downstream(t4)
t2.set_downstream(t5)
t2.set_downstream(t6)
t3.set_downstream(t4)
t3.set_downstream(t5)
t3.set_downstream(t6)
```

Parameters

- **from_tasks** (*List[airflow.models.BaseOperator]*) – List of tasks to start from.
- **to_tasks** (*List[airflow.models.BaseOperator]*) – List of tasks to set as downstream dependencies.

---

class **airflow.models.baseoperator.BaseOperatorLink**[source]

Abstract base class that defines how we get an operator link.

operators *:ClassVar[List[Type[BaseOperator]]] = []* [source]

This property will be used by Airflow Plugins to find the Operators to which you want to assign this Operator Link

Returns

List of Operator classes used by task for which you want to create extra link

name[source]

Name of the link. This will be the button name on the task UI.

Returns

link name

get_link(*self*, *operator: BaseOperator*, *dttm: datetime*)

[source]

Link to external system.

Parameters

- **operator** – airflow operator
- **dttm** – datetime

Returns

link to external system

Previous Next