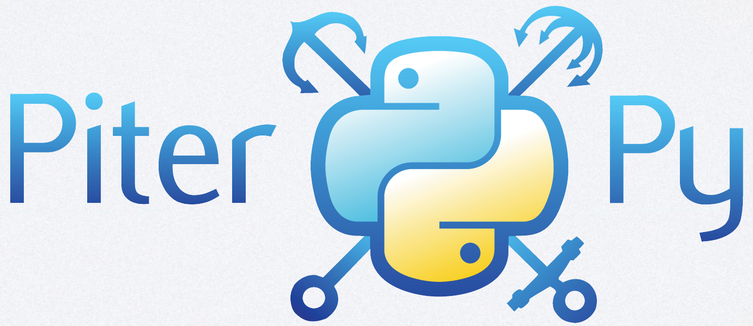
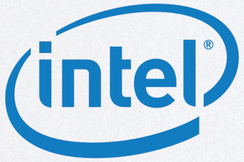
DEVELOPING ELEGANT **WORKFLOWS**

with **Apache Airflow**

Michał Karzyński • PiterPy #5



ABOUT ME

- **Michał Karzyński** (@postrational)
- Full stack geek (**Python**, **JavaScript** and **Linux**)
- I **blog** at http://michal.karzynski.pl
- I'm a **tech lead** at  and a **consultant** at  **ATARNIA**.com

LET'S TALK ABOUT **WORKFLOWS**

WHAT IS A **WORKFLOW?**

- sequence of **tasks**
- started on a **schedule** or **triggered** by an event
- frequently used to handle big **data processing pipelines**

A TYPICAL **WORKFLOW**

EXAMPLES EVERYWHERE

- Extract, Transform, Load (ETL)
- data warehousing
- A/B testing
- anomaly detection
- training recommender systems
- orchestrating automated testing
- processing genomes every time a new genome file is published

WORKFLOW **MANAGERS**

Oozie

Luigi

Airflow

Azkaban

Taskflow

APACHE **AIRFLOW**

- **open source**, written in Python
- developed originally by **Airbnb**

Apache Airflow

- **280+ contributors**, 4000+ commits, 5000+ stars
- used by **Intel**, Airbnb, Yahoo, PayPal, WePay, Stripe, Blue Yonder…

APACHE **AIRFLOW**

1. **Framework** to write your workflows
2. Scalable **executor** and **scheduler**
3. Rich **web UI** for monitoring and logs

Apache Airflow

Demo

WHAT **FLOWS** IN A WORKFLOW?

photo by Steve Byrne

WHAT **FLOWS** IN A WORKFLOW?

Tasks make decisions based on:

- workflow input
- upstream task output

photo by Steve Byrne

WHAT **FLOWS** IN A WORKFLOW?

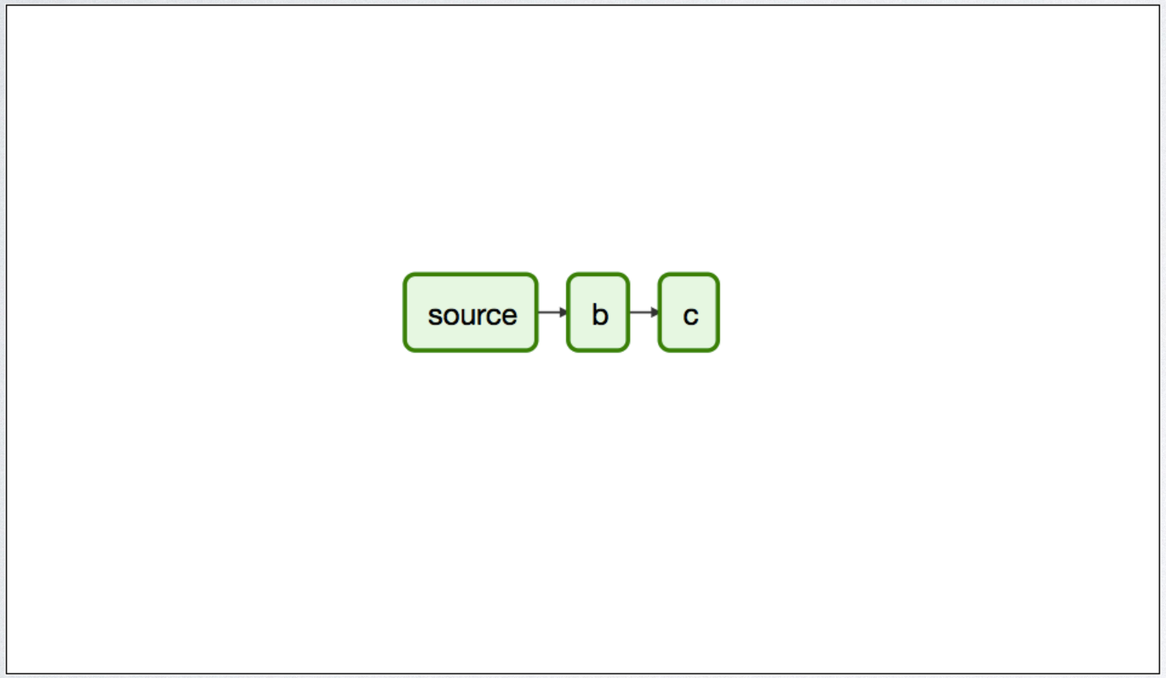
Tasks make decisions based on:

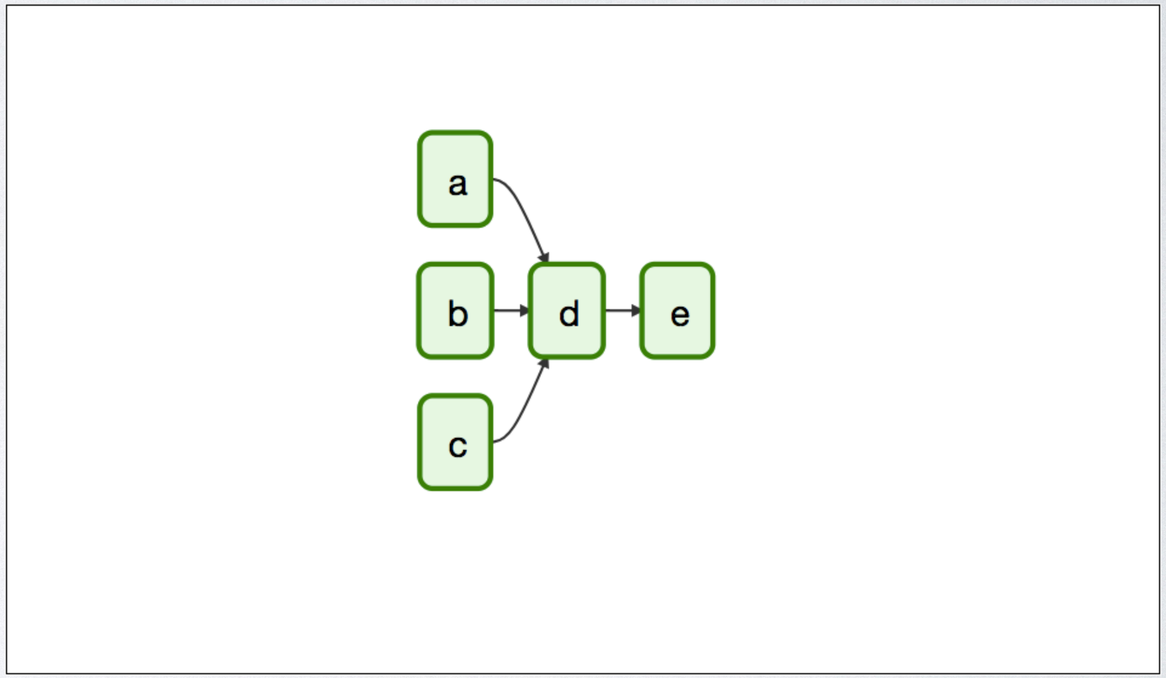
- workflow input
- upstream task output

Information flows downstream like a **river**.

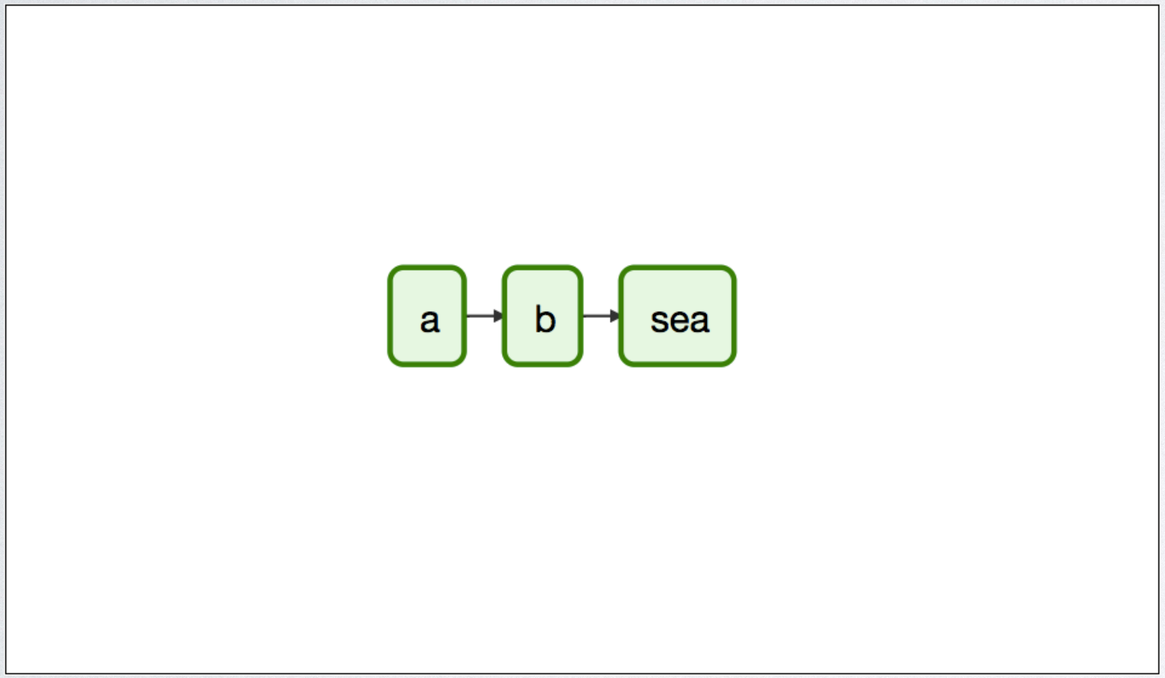
photo by Steve Byrne

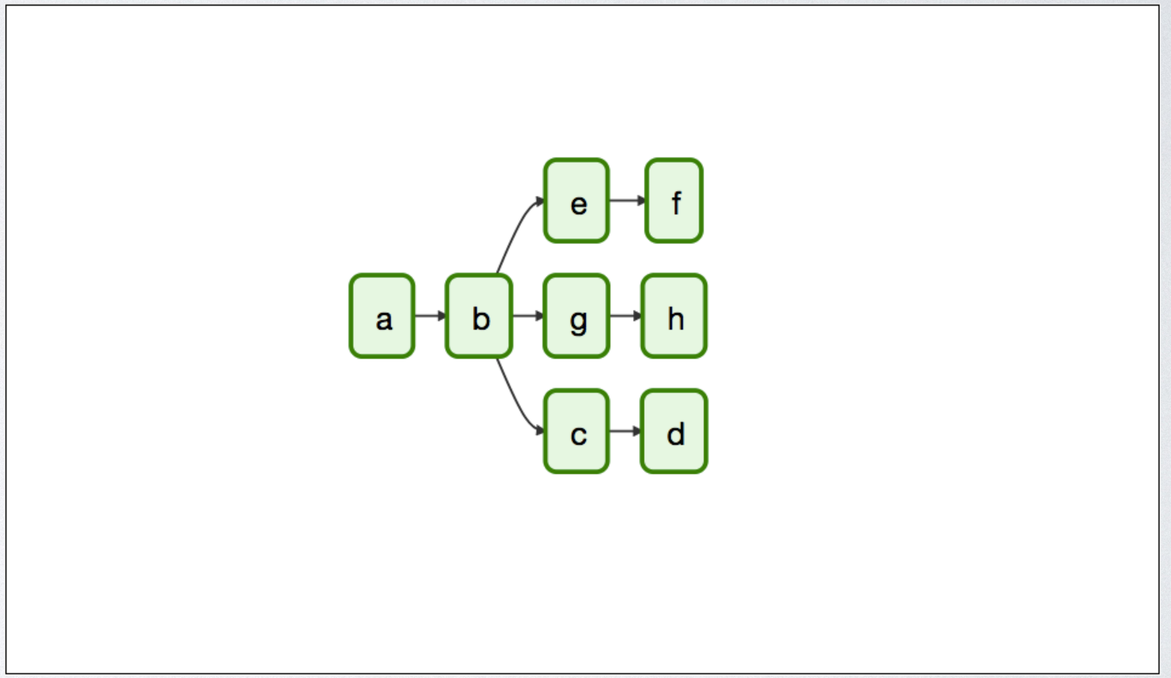
SOURCE AND TRIBUTARIES



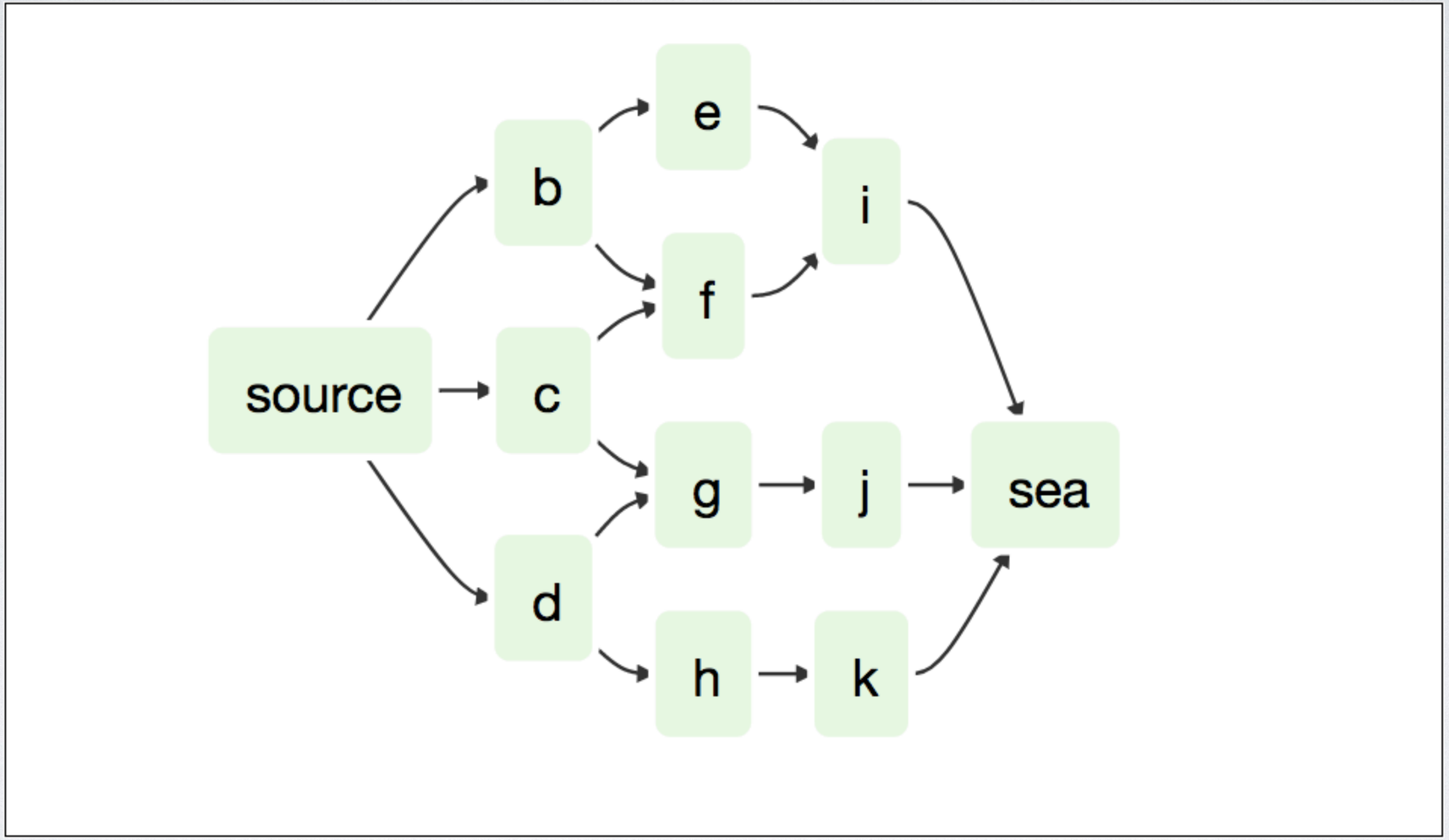


DISTRIBUTARIES AND DELTAS

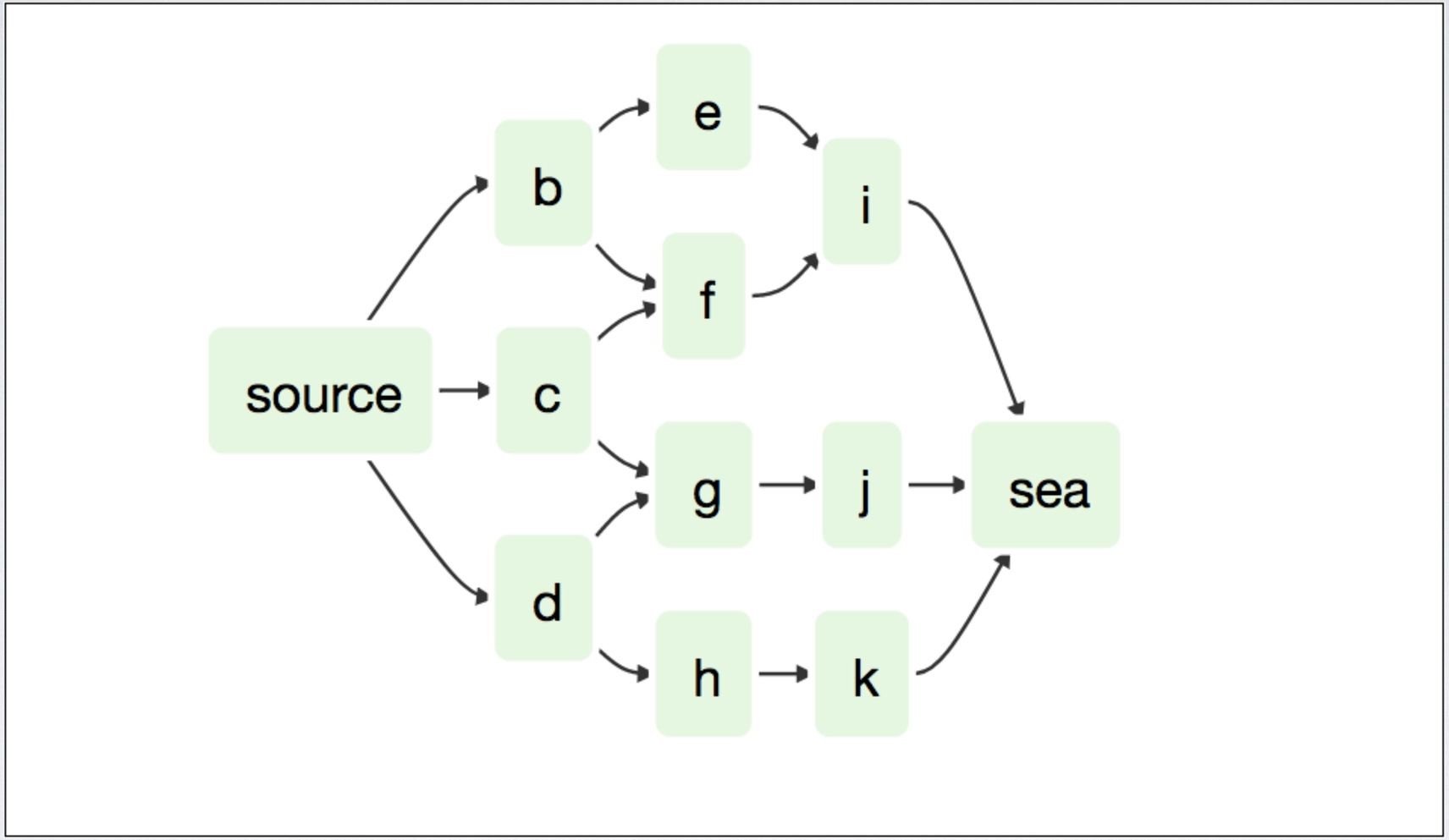




BRANCHES?

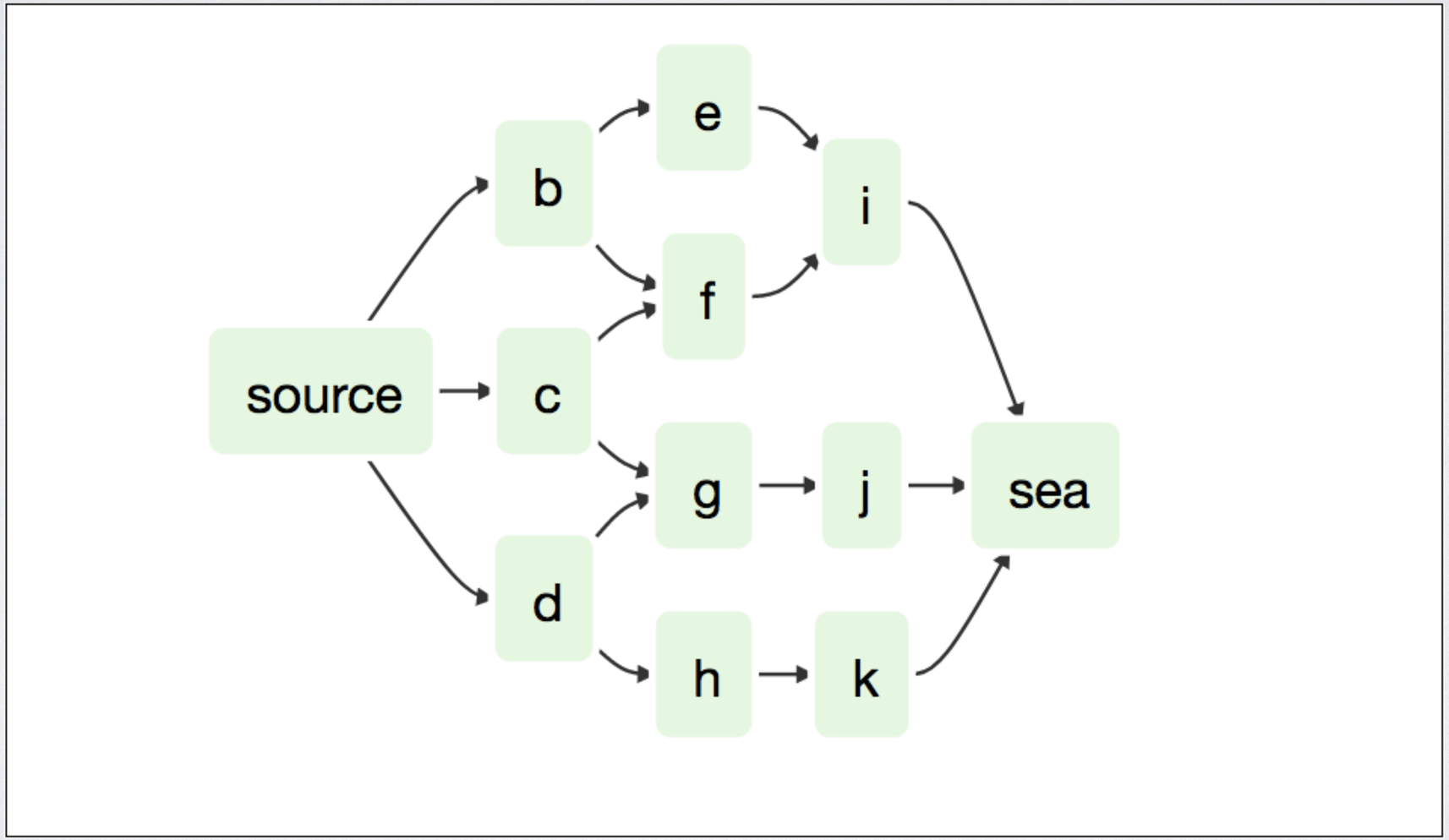


BRANCHES?

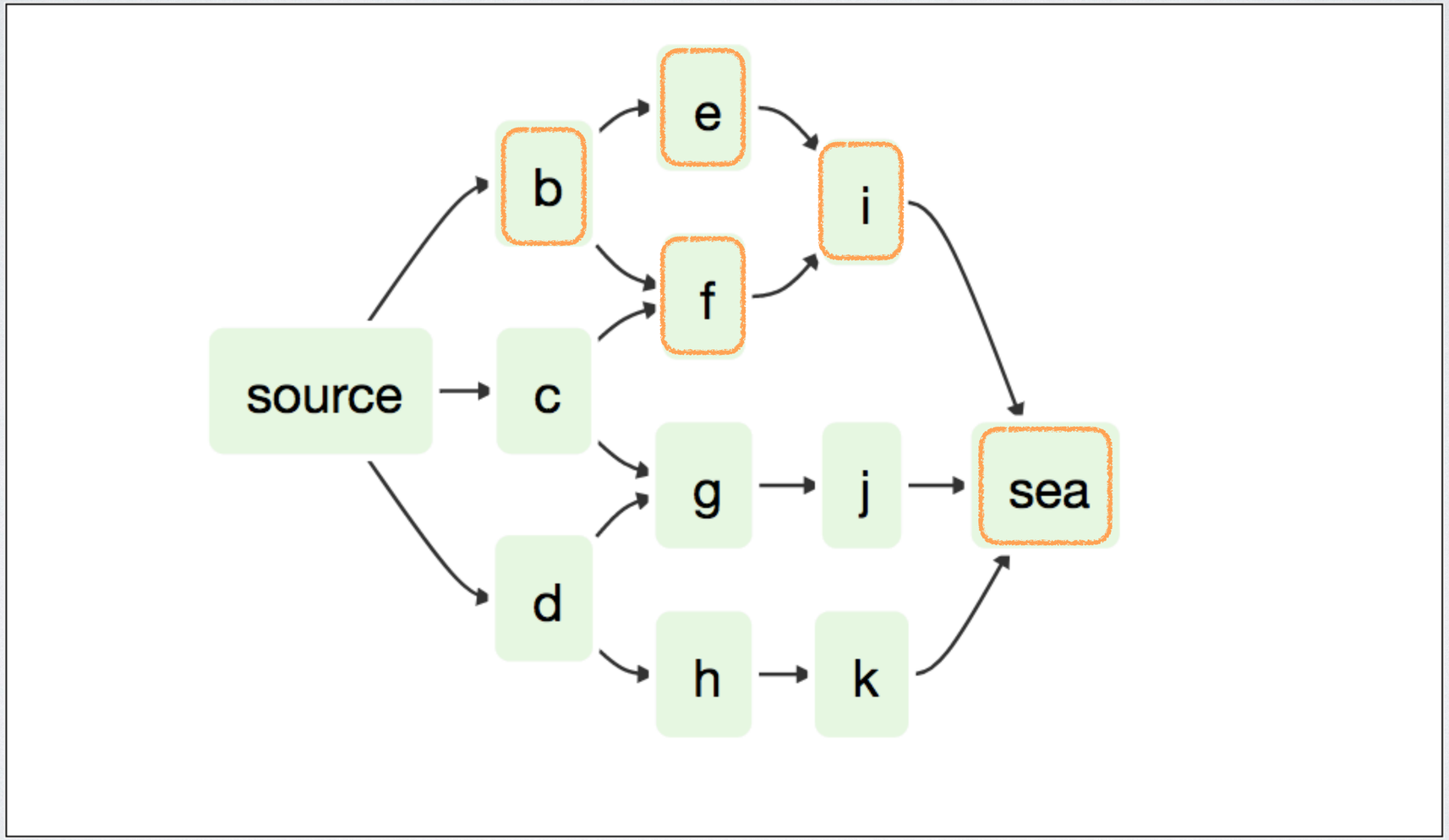


Directed Acyclic Graph (**DAG**)

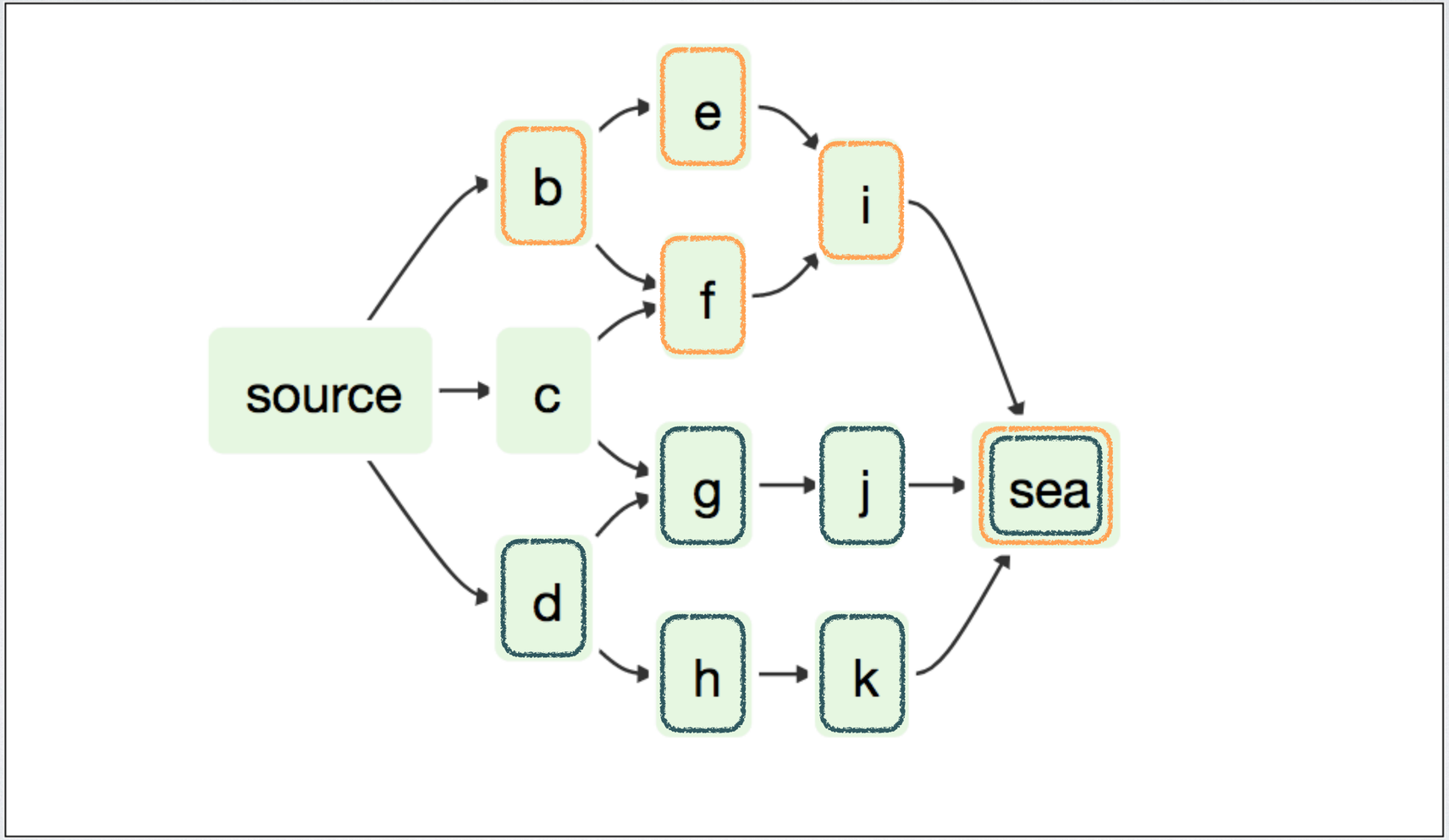
FLOW



FLOW



FLOW



AIRFLOW CONCEPTS: **DAGS**

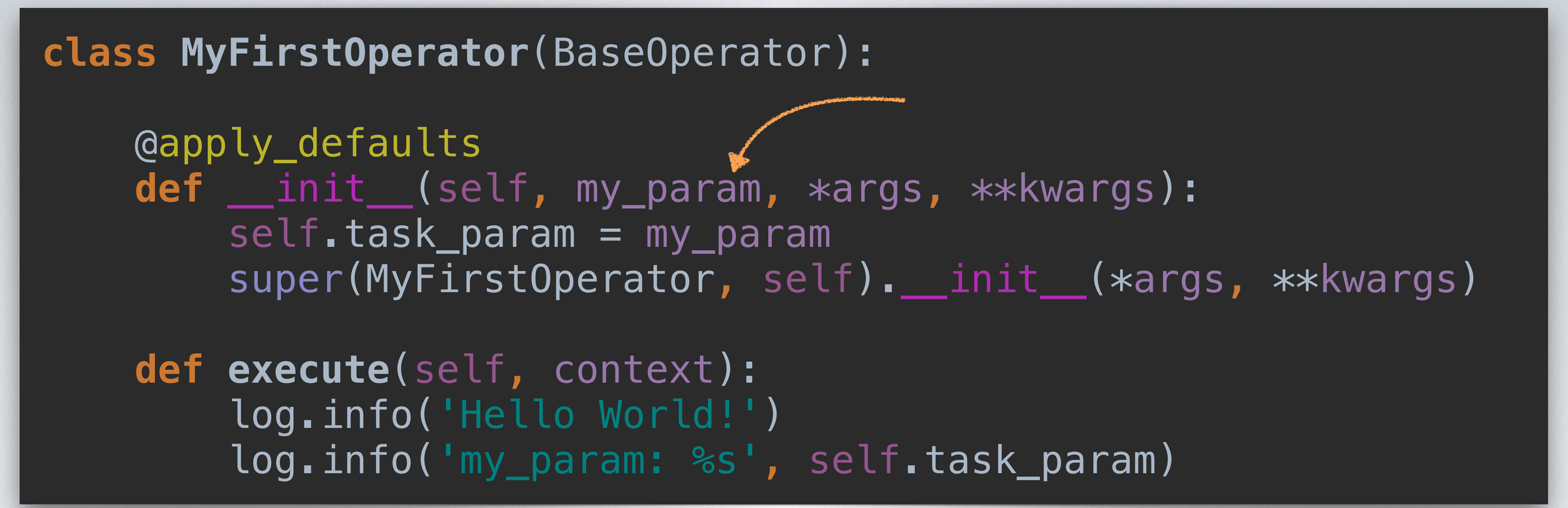
- DAG - **Directed Acyclic Graph**
- Define workflow **logic as shape** of the graph

```
def print_hello():
    return 'Hello world!'

dag = DAG('hello_world', description='Simple tutorial DAG',
          schedule_interval='0 12 * * *',
          start_date=datetime.datetime(2018, 11, 3), catchup=False)

with dag:

    dummy_task = DummyOperator(task_id='dummy', retries=3)

    hello_task = PythonOperator(task_id='hello', python_callable=print_hello)

    dummy_task >> hello_task
```

AIRFLOW CONCEPTS: **OPERATOR**

- definition of a **single task**
- will **retry** automatically
- should be **idempotent**
- Python class with an `execute` method

```python
class MyFirstOperator(BaseOperator):

    @apply_defaults
    def __init__(self, my_param, *args, **kwargs):
        self.task_param = my_param
        super(MyFirstOperator, self).__init__(*args, **kwargs)

    def execute(self, context):
        log.info('Hello World!')
        log.info('my_param: %s', self.task_param)
```




```
with dag:
    my_first_task = MyFirstOperator(my_param='This is a test.',
                                    task_id='my_task')
```

AIRFLOW CONCEPTS: **SENSORS**

- **long running** task
- useful for **monitoring** external processes
- Python class with a **`poke`** method
- **`poke`** will be called repeatedly until it returns **`True`**

```python
class MyFirstSensor(BaseSensorOperator):

    def poke(self, context):
        current_minute = datetime.now().minute
        if current_minute % 3 != 0:
            log.info('Current minute (%s) not is divisible by 3, '
                     'sensor will retry.', current_minute)
            return False

        log.info('Current minute (%s) is divisible by 3, '
                 'sensor finishing.', current_minute)
        task_instance = context['task_instance']
        task_instance.xcom_push('sensors_minute', current_minute)
        return True
```

AIRFLOW CONCEPTS: **XCOM**

- means of **communication** between task instances
- saved in **database** as a pickled object
- best suited for **small** pieces of data (ids, etc.)

XCom **Push**:

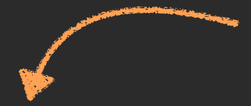
```
def execute(self, context):
    ...
    task_instance = context['task_instance']
    task_instance.xcom_push('sensors_minute', current_minute)
```

XCom **Pull**:

```
def execute(self, context):
    ...
    task_instance = context['task_instance']
    sensors_minute = task_instance.xcom_pull('sensor_task_id', key='sensors_minute')
    log.info('Valid minute as determined by sensor: %s', sensors_minute)
```

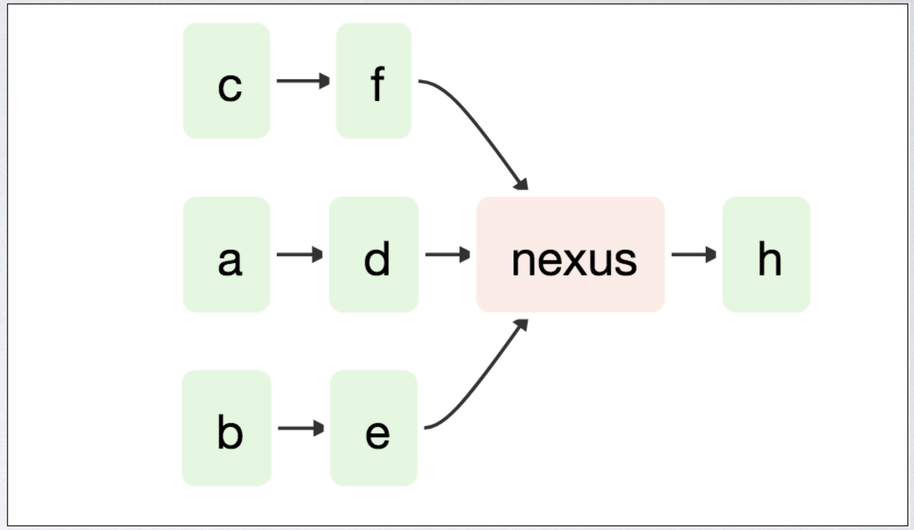
SCAN FOR INFORMATION UPSTREAM

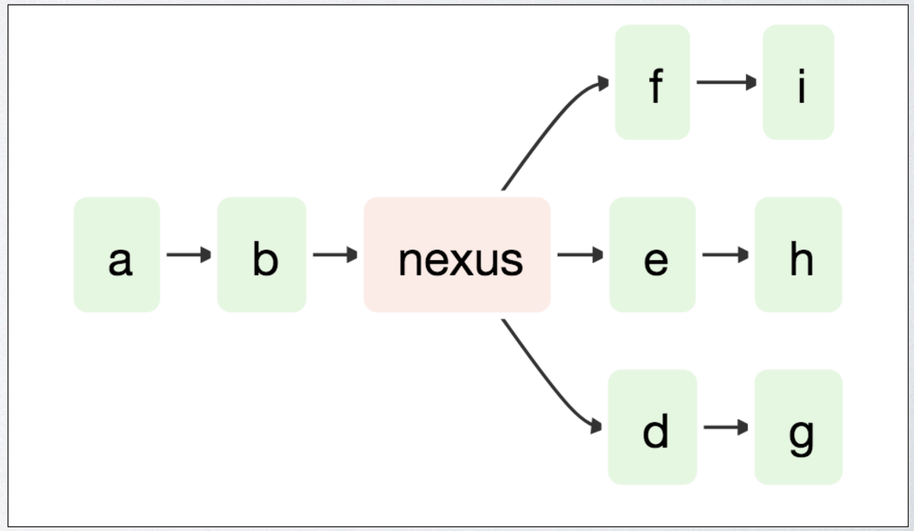
```
def execute(self, context):
    log.info('XCom: Scanning upstream tasks for Database IDs')

    task_instance = context['task_instance']

    upstream_tasks = self.get_flat_relatives(upstream=True)
    upstream_task_ids = [task.task_id for task in upstream_tasks]
    upstream_database_ids = task_instance.xcom_pull(task_ids=upstream_task_ids, key='db_id')

    log.info('XCom: Found the following Database IDs: %s', upstream_database_ids)
```



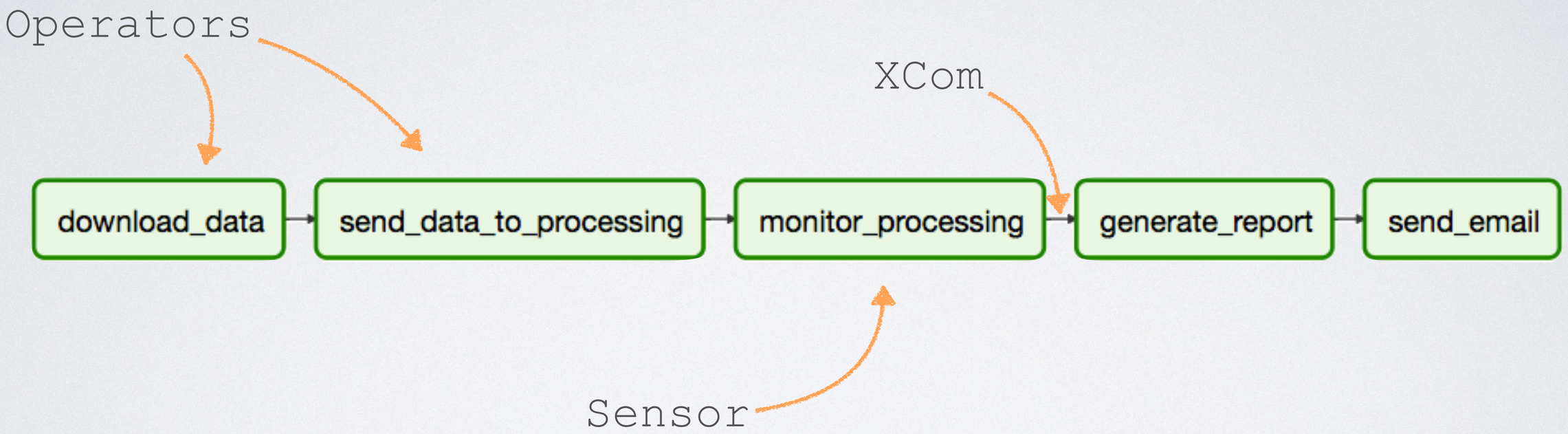
REUSABLE OPERATORS

- **loosely coupled**
- with **few necessary XCom parameters**
- **most** parameters are **optional**
- **sane defaults**
- will **adapt** if information appears upstream





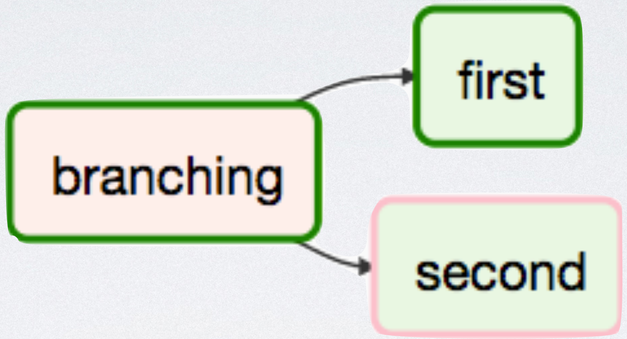
A TYPICAL **WORKFLOW**



CONDITIONAL EXECUTION: **BRANCH OPERATOR**

- decide **which branch** of the graph to follow
- all others will be **skipped**

CONDITIONAL EXECUTION:
BRANCH OPERATOR




```
def choose():
    return 'first'

with dag:
    branching = BranchPythonOperator(task_id='branching', python_callable=choose)
    branching >> DummyOperator(task_id='first')
    branching >> DummyOperator(task_id='second')
```

CONDITIONAL EXECUTION: AIRFLOW SKIP EXCEPTION

```
    def execute(self, context):
        ...
        if not conditions_met:
            log.info('Conditions not met, skipping.')
            raise AirflowSkipException()
```

- raise `AirflowSkipException` to skip execution of current task
- all other exceptions cause retries and ultimately the task to fail
- puts a **dam** in the river

CONDITIONAL EXECUTION: TRIGGER RULES

- decide when a task is triggered
- defaults to `all_success`
- `all_done` - **opens dam** from downstream task

```
class TriggerRule(object):
    ALL_SUCCESS = 'all_success'
    ALL_FAILED = 'all_failed'
    ALL_DONE = 'all_done'
    ONE_SUCCESS = 'one_success'
    ONE_FAILED = 'one_failed'
    DUMMY = 'dummy'
```

BASH COMMANDS AND **TEMPLATES**

- execute **Bash** command on Worker node
- use **Jinja** templates to generate a Bash script
- define **macros** - Python functions used in templates

BASH COMMANDS AND **TEMPLATES**

```
templated_command = """
    {% for i in range(5) %}
        echo "execution date: {{ ds }}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Value I passed in'},
    dag=dag)
```

AIRFLOW **PLUGINS**

- Add many types of components used by Airflow
- Subclass of `AirflowPlugin`
- File placed in `AIRFLOW_HOME/plugins`

AIRFLOW **PLUGINS**

```
class MyPlugin(AirflowPlugin):
    name = "my_plugin"

    # A list of classes derived from BaseOperator
    operators = []

    # A list of menu links (flask_admin.base.MenuLink)
    menu_links = []

    # A list of objects created from a class derived from flask_admin.BaseView
    admin_views = []

    # A list of Blueprint object created from flask.Blueprint
    flask_blueprints = []

    # A list of classes derived from BaseHook (connection clients)
    hooks = []

    # A list of classes derived from BaseExecutor (e.g. MesosExecutor)
    executors = []
```

Introductory Airflow tutorial available on my blog:

michal.karzynski.pl

Introductory Airflow tutorial available on my blog:

michal.karzynski.pl

THANK YOU