Class of 04/29/2019

Airflow DAG Layout

- start date - the date specifying when the workflow will be run since
- name - the unique identifier for each DAG so Airflow knows which DAG to invoke
- schedule interval - the time interval that marks when the Airflow workflow will be run again

Airflow Code for Humans: Understanding Workflows By Example

Using workflow files from the snippets wiki, we can walk through what each line means.

simple_workflow1.py

```
import datetime
from airflow import models
from airflow.operators.bash_operator import BashOperator
```

Various imports required by Airflow. Generally doesn't need to be changed for the intents of our class.

```
default_dag_args = {
    # https://airflow.apache.org/faq.html#what-s-the-deal-with-start-date
    'start_date': datetime.datetime(2019, 4, 1)
}
```

Here we specify the start date for when we would like our workflow to start running.

```
sql1='create table college.Class2 as select * from college.Class'
sql2='create table college.Class3 as select count(*) as count from college.Class2'
```

Here we specify some strings we will be using in the actual DAG below. It's not necessary that you specify these strings exactly here, but Shirley chooses to so her code is cleaner.

```
with models.DAG(
        'simple_workflow1',
        schedule_interval=datetime.timedelta(days=1),
        default_args=default_dag_args) as dag:
```

We begin to run our DAG that we specify in the body here. Notice we pass in the name and schedule interval of our DAG, in addition to passing the start date that we declared earlier.

```
    copy_class_table = BashOperator(
            task_id='copy_class_table',
            bash_command='bq query --use_legacy_sql=false "' + sql1 + '"')

    get_class_count = BashOperator(
            task_id='get_class_count',
            bash_command='bq query --use_legacy_sql=false "' + sql2 + '"') # default trigger
rule is all_success
```

We declare tasks in our workflow using "BashOperator"s, which consists of a task ID and the command to run. They are called BashOperators because they are essentially commands to be run in the Bash shell.

```
    copy_class_table >> get_class_count
```

This final line specifies that the task `copy_class_table` should run, and then `get_class_count`.

You may remember that the operator >> in Apache Beam was overridden to feed a PCollection into a PTransform. Now, Airflow has overridden the operator to mean "run this, *then* this" when applied to Airflow tasks.

`simple_workflow2.py` adds what are called "trigger rules" to tasks, which lets Airflow know that this task should only execute if this trigger is met. From the file:

```
    get_class_count_create = BashOperator(
            task_id='get_class_count_create',
            bash_command='bq query --use_legacy_sql=false "' + sql1 + '"',
            trigger_rule='all_done') # trigger only if parent task has completed
```

This operator is only run if all previous tasks are run, whether or not they succeeded because we specified the `'all_done'` trigger rule.

`college_workflow2.py` shows a new feature of Airflow. Looking at the last line,

```
delete_dataset >> create_dataset >> [create_student_table, create_teacher_table,
create_class_table, create_teaches_table, create_takes_table]
```

notice that the final task is actually a list of tasks. Putting tasks in a list means that all tasks in the list will be running concurrently.

Be careful -- make sure it's true that *all* of these tasks can run concurrently and have no dependencies with one another, because there is *no guarantee* as to what order they will be run in. As programming students, parallel computing is something you should be very, VERY careful with because it's hard to debug and is very common.