ploomber

Release 0.23.3

ploomber

Sep 18, 2024

CONTENTS

New to Ploomber? Start here:

CHAPTER

ONE

SECTIONS

Get Started	High-level tutorials covering the basics of Ploomber.
Use Cases	High-level descriptions of what you can build with Ploomber.
User Guide	In-depth tutorials for developing Ploomber pipelines.
Deployment	In-depth tutorials for deployment.
Cookbook	Quick reference for common patterns.
API Reference	Technical documentation.
Community	General information about our community and the project.

CHAPTER

TWO

INDEX

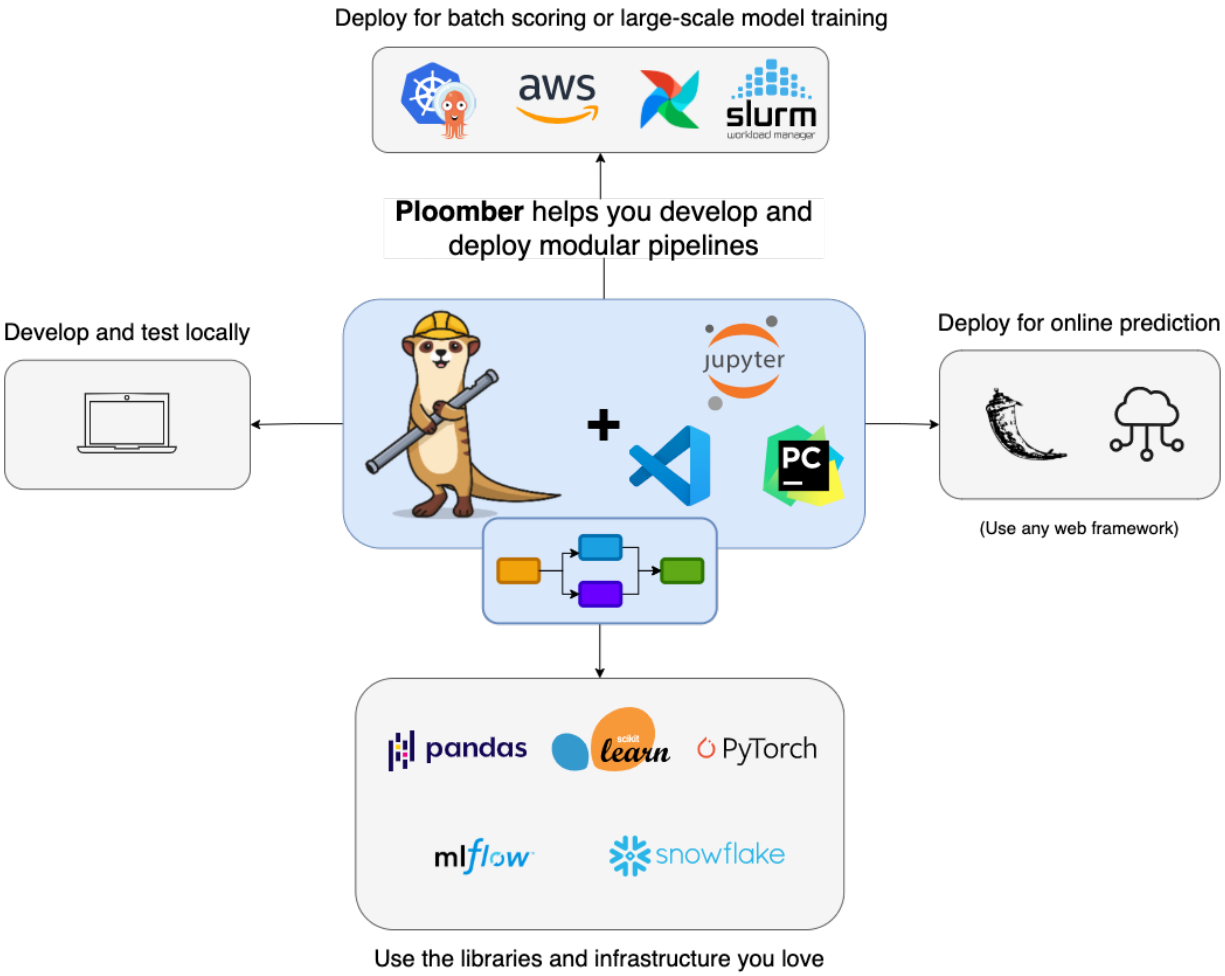
2.1 Get Started

2.1.1 What is Ploomber?

Ploomber is a framework to build collaborative and modular pipelines; it integrates with Jupyter but you can use it with any other editor.

Ploomber eliminates the *notebook refactoring problem*: data teams prototype their work in Jupyter notebooks and then refactor the code for deployment. Such refactoring process comes with a high risk, since it steeply increases the chance of breaking the analysis when moving the code around, and slows down progress.

With Ploomber, you can develop maintainable, collaborative, and production-ready pipelines from day one.



Tell me more

- Watch this 6-minute video to see how the experience looks like.
- Read about *common use cases*.
- Check out the Videos section to watch some of our presentations.
- Read our guest blog post in the official Jupyter blog to learn more about our mission.

I have questions

- Join our community.
- Send us an email (contact@ploomber.io).
- Open an issue on GitHub.

I'm ready to start building!

- Head over to our *first tutorial* for a short walkthrough to run your first Ploomber pipeline.
- Check out our *Downloading templates* guide to run some pre-configured examples.

2.1.2 Quickstart

pip

```
pip install ploomber
```

conda

```
conda install ploomber -c conda-forge
```

What's next?

- **Bring your own code!** Check out the tutorial to migrate your code into Ploomber: *Refactoring legacy notebooks*.
- Check the **introductory tutorial**: *Your first Python pipeline*.
- Run more *examples*.

To run this locally, install Ploomber and execute: `ploomber examples -n guides/first-pipeline`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.1.3 Your first Python pipeline

Introductory tutorial to learn the basics of Ploomber.

Introduction

Ploomber helps you build modular pipelines. A pipeline (or **DAG**) is a group of tasks with a particular execution order, where subsequent (or **downstream** tasks) use previous (or **upstream**) tasks as inputs.

Pipeline declaration

This example pipeline contains five tasks, `1-get.py`, `2-profile-raw.py`, `3-clean.py`, `4-profile-clean.py` and `5-plot.py`; we declare them in a `pipeline.yaml` file:

```
# Content of pipeline.yaml
tasks
   # source is the code you want to execute (.ipynb also supported)
    source  1-get.py
    # products are task's outputs
    product
```

(continues on next page)

(continued from previous page)

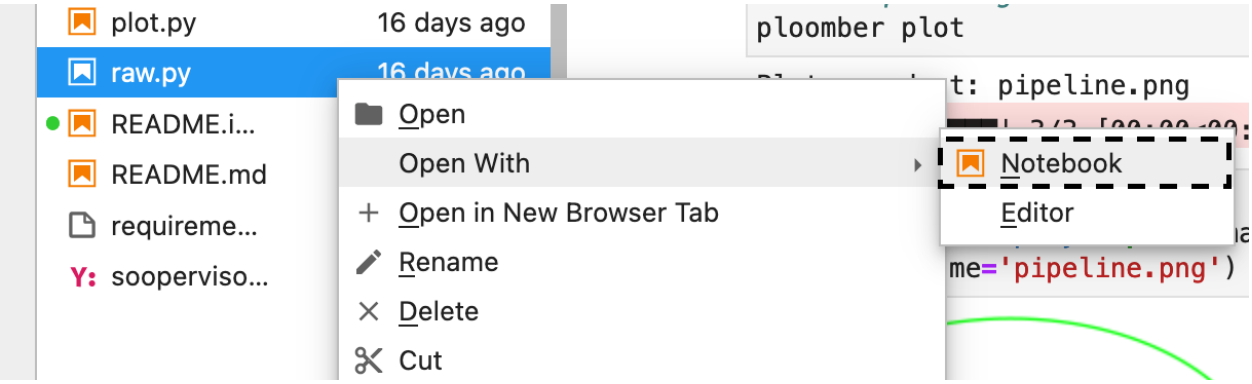
```
    # scripts generate executed notebooks as outputs
    nb  output/1-get.html
    # you can define as many outputs as you want
    data  output/raw_data.csv

  source  2-profile-raw.py
  product  output/2-profile-raw.html

  source  3-clean.py
  product
    nb  output/3-clean.html
    data  output/clean_data.parquet

  source  4-profile-clean.py
  product  output/4-profile-clean.html

  source  5-plot.py
  product  output/5-plot.html
```

Note: YAML is a human-readable text format similar to JSON.

Note: Ploomber supports Python scripts, Python functions, Jupyter notebooks, R scripts, and SQL scripts.

Opening `.py` files as notebooks

Ploomber integrates with Jupyter. Among other things, it **allows you to open ``.py`` files as notebooks** (via `jupytext`).



What sets the execution order?

Ploomber infers the pipeline structure from your code. For example, to clean the data, we must get it first; hence, we declare the following in `3-clean.py`:

```
# 3-clean.py

# this tells Ploomber to execute the '1-get' task before '3-clean'
        =  '1-get'
```

Plotting the pipeline

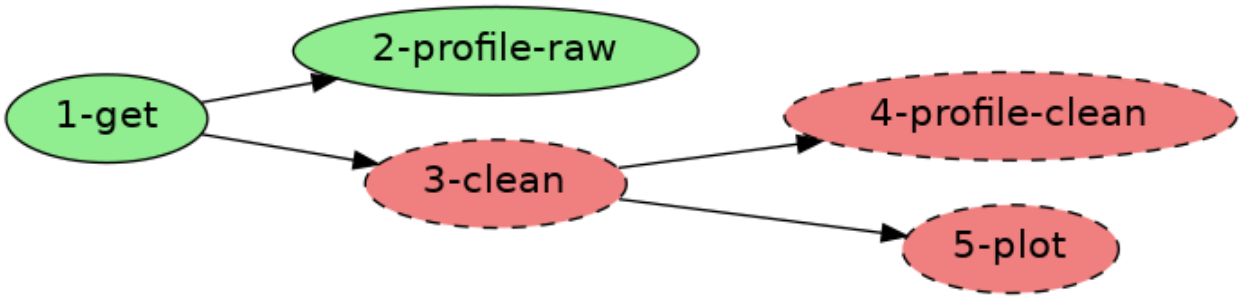
[1]:
```
%%bash
ploomber plot
```

```
Loading pipeline...
Plot saved at: pipeline.png
```

```
100%|| 5/5 [00:00<00:00, 15477.14it/s]
```

[2]:
```
from                 import
              ='pipeline.png'
```

[2]:



You can see that our pipeline has a defined execution order.

Note: This is a sample predefined five-task pipeline, Ploomber can manage arbitrarily complex pipelines and dependencies among tasks.

Running the pipeline

[3]:
```
%%bash
# takes a few seconds to finish
ploomber build
```

```
Loading pipeline...
name             Ran?      Elapsed (s)    Percentage
---------------  ------  -------------  ------------
3-clean          True          5.43922       28.676
4-profile-clean  True          4.76502       25.1215
5-plot           True          8.76362       46.2025
1-get            False         0              0
2-profile-raw    False         0              0
```

```
Building task '3-clean':   0%|          | 0/3 [00:00<?, ?it/s]
Executing:   0%|          | 0/9 [00:00<?, ?cell/s]
Executing:  11%|         | 1/9 [00:01<00:11,  1.46s/cell]
Executing: 100%|| 9/9 [00:05<00:00,  1.79cell/s]
Building task '4-profile-clean':  33%|      | 1/3 [00:05<00:10,  5.44s/it]
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
Executing:  14%|        | 1/7 [00:01<00:06,  1.04s/cell]
Executing:  43%|     | 3/7 [00:02<00:03,  1.25cell/s]
Executing:  71%|  | 5/7 [00:03<00:01,  1.58cell/s]
Executing:  86%| | 6/7 [00:03<00:00,  1.87cell/s]
Executing: 100%|| 7/7 [00:04<00:00,  1.59cell/s]
```

(continues on next page)

(continued from previous page)

```
Building task '5-plot':  67%|   | 2/3 [00:10<00:05,  5.04s/it]
Executing:   0%|          | 0/8 [00:00<?, ?cell/s]
Executing:  12%|        | 1/8 [00:02<00:17,  2.46s/cell]
Executing:  62%|   | 5/8 [00:02<00:01,  2.18cell/s]
Executing:  75%|  | 6/8 [00:05<00:01,  1.04cell/s]
Executing: 100%|| 8/8 [00:08<00:00,  1.05s/cell]
Building task '5-plot': 100%|| 3/3 [00:18<00:00,  6.32s/it]
```

This pipeline saves all the output in the `output/` directory; we have the output notebooks and data files:

[4]:
```
%%bash
ls output
```

```
1-get.html
2-profile-raw.html
3-clean.html
4-profile-clean.html
5-plot.html
clean_data.parquet
raw_data.csv
```

Updating the pipeline

Ploomber automatically caches your pipeline's previous results and only runs tasks that changed since your last execution.

Execute the following to modify the `3-clean.py` script

[5]:
```
from          import

     =       '3-clean.py'
      =      .

# add a print statement at the end of 3-clean.py
    .                  + """
print("hello")
"""
```

[5]:
```
397
```

Execute the pipeline again:

[6]:
```
%%bash
# takes a few seconds to finish
ploomber build
```

```
Loading pipeline...
name             Ran?      Elapsed (s)    Percentage
---------------  ------  -------------  ------------
3-clean          True          2.22814       14.2248
4-profile-clean  True          4.7084        30.0591
5-plot           True          8.72726       55.7161
1-get            False         0              0
2-profile-raw    False         0              0
```

```
Building task '3-clean':   0%|          | 0/3 [00:00<?, ?it/s]
Executing:   0%|          | 0/9 [00:00<?, ?cell/s]
Executing:  11%|         | 1/9 [00:01<00:10,  1.29s/cell]
Executing: 100%|| 9/9 [00:01<00:00,  5.04cell/s]
Building task '4-profile-clean':  33%|      | 1/3 [00:02<00:04,  2.23s/it]
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
Executing:  14%|        | 1/7 [00:00<00:05,  1.10cell/s]
Executing:  43%|     | 3/7 [00:02<00:03,  1.23cell/s]
Executing:  71%|  | 5/7 [00:03<00:01,  1.58cell/s]
Executing:  86%| | 6/7 [00:03<00:00,  1.91cell/s]
Executing: 100%|| 7/7 [00:04<00:00,  1.60cell/s]
Building task '5-plot':  67%|   | 2/3 [00:06<00:03,  3.69s/it]
Executing:   0%|          | 0/8 [00:00<?, ?cell/s]
Executing:  12%|        | 1/8 [00:02<00:16,  2.39s/cell]
Executing:  62%|   | 5/8 [00:02<00:01,  2.24cell/s]
Executing:  75%|  | 6/8 [00:05<00:01,  1.03cell/s]
Executing: 100%|| 8/8 [00:08<00:00,  1.05s/cell]
Building task '5-plot': 100%|| 3/3 [00:15<00:00,  5.22s/it]
```

[7]:
```
# restore contents
    .
```

[7]:
```
381
```

You'll see that `1-get.py` & `2-profile-raw.py` didn't run because it was not affected by the change!

Where to go from here

Bring your own code! Check out the tutorial to migrate your code to Ploomber.

Have questions? Ask us anything on Slack.

Want to dig deeper into Ploomber's core concepts? Check out the basic concepts tutorial.

Want to start a new project quickly? Check out how to get examples.

2.1.4 Basic concepts

This guide explains Ploomber's core concepts.

Ploomber allows you to quickly turn a collection of scripts, notebooks, or functions into a data pipeline by following three conventions:

1. Each task is a function, script or notebook.
2. Tasks declare their dependencies using an `upstream` variable.
3. Tasks declare their outputs using a `product` variable.

A simple pipeline

Let's say we want to build a pipeline to plot some data. Instead of coding everything in a single file, we'll break down logic in three steps, which will make our code more maintainable and easier to test:

Note: A pipeline is also known as a directed acyclic graph (or DAG). We use both of these terms interchangeably.

In a Ploomber pipeline, outputs (also known as **products**) from one task become inputs of "downstream" tasks. Hence, "upstream" dependencies read from left to right. For example, `raw` is an "upstream" dependency of `clean`.

An "upstream" dependency implies that a given task uses its upstream products as inputs. Following the pipeline example, `clean` uses `raw`'s products, and `plot` uses `clean`'s products.

Ploomber supports three types of tasks:

1. Python functions (also known as callables)
2. Python scripts/notebooks (and their R equivalents)
3. SQL scripts

You can mix any combination of tasks in your pipeline. For example, you can dump data with a SQL query, then plot it with Python.

Defining a pipeline

To execute a pipeline, Ploomber needs to know the location of the task's source code (`source` key), and the location of the task's products (`product` key). You can do this via a `pipeline.yaml` file:

```
tasks
  # this is a sql script task
    source  raw.sql
    product
    # ...

  # this is a function task
  # "my_functions.clean" is equivalent to: from my_functions import clean
    source  my_functions.clean
    product  output/clean.csv

  # this is a script task (notebooks work the same)
    source  plot.py
    product
      # scripts always generate a notebook (more on this in the next section)
      nb  output/plots.ipynb
```

Note: You can set a task name using `name`. If not present, Ploomber infers it from the `source` value by removing the extension to the file's name.

Once you have a `pipeline.yaml` file, you can run it with:

```
ploomber build
```

Ploomber keeps track of source changes to skip up-to-date tasks. If you run `ploomber build` again, only tasks whose source code has changed are executed. This helps iterate faster, as changes to the pipeline only trigger the least number of tasks.

Tip: You can use the `resources_` section in a task definition to tell Ploomber to track the content of other files. *Click here to learn more*.

For a full reference on `pipeline.yaml` files see: *Spec API (pipeline.yaml)*.

Let's now see how to use scripts and notebooks as pipeline tasks.

Tasks: scripts/notebooks

Jupyter notebooks files (`.ipynb`) contain both code and output; while convenient, keeping code and outputs in the same file makes version control (i.e., `git`) difficult.

Our recommended approach is to use scripts as sources. However, thanks to the integration with Jupyter, **you can open scripts as notebooks**. The following image shows a side-by-side comparison of the same source code as `.py` (script) and as a `.ipynb` (notebook) file:

clean.py

```python
# %%
import pandas as pd

# %% tags=["parameters"]
upstream = ['raw']
product = None

# %%
df = pd.read_csv(upstream['raw']['data'])
# some data cleaning code...

# %%
df.to_csv(product['data'], index=False)

```

clean.ipynb

```
[ ]: import pandas as pd

[ ]: upstream = ['raw']
     product = None

[ ]: df = pd.read_csv(upstream['raw']['data'])
     # some data cleaning code...

[ ]: df.to_csv(product['data'], index=False)
```

Note that the `.py` script has some `# %%` comments. Such markers allow us to delimit code cells and render the `.py` file as a notebook.

Note: The `# %%` is one way of representing `.py` as notebooks. Ploomber uses `jupytext` to perform the conversion, other formats such as the "light" (`# +`) format work too. Editors such as VS Code, Spyder, and PyCharm support the "percent" format.

To keep the benefits of the `.ipynb` format, **Ploomber creates a copy of your scripts, converts them to .ipynb at runtime and executes them.** This is a crucial concept: scripts are part of your project's source code, but executed notebooks are pipeline products.

Note: Even though we recommend the use of `.py` files, you can still use regular `.ipynb` files as sources if you prefer so.

To know more about integration with Jupyter notebooks, see the *Jupyter integration* guide.

R scripts/notebooks are supported as well.

`upstream` and `product`

To specify task dependencies, include a special `parameters` cell in your script/notebook. Following the example pipeline, `clean` has `raw` as an upstream dependency as the **raw** task is an input to the **clean** task. We establish this relation by declaring an `upstream` variable with a list of task names that should execute **before** the file we're editing. If a script/notebook has no dependencies, set `upstream = None`.

```
# %% tags=["parameters"]
         =  'raw'  # this means: execute raw.py, then clean.py
        = None
```

Important: `product = None` is a placeholder. It states that our script takes an input parameter called `product`, but the actual value is automatically replaced at runtime, we explain this in the upcoming section.

Note: the `# %%` markers only apply to scripts. Click here for information on adding tags to `.ipynb` files.

The cell injection process

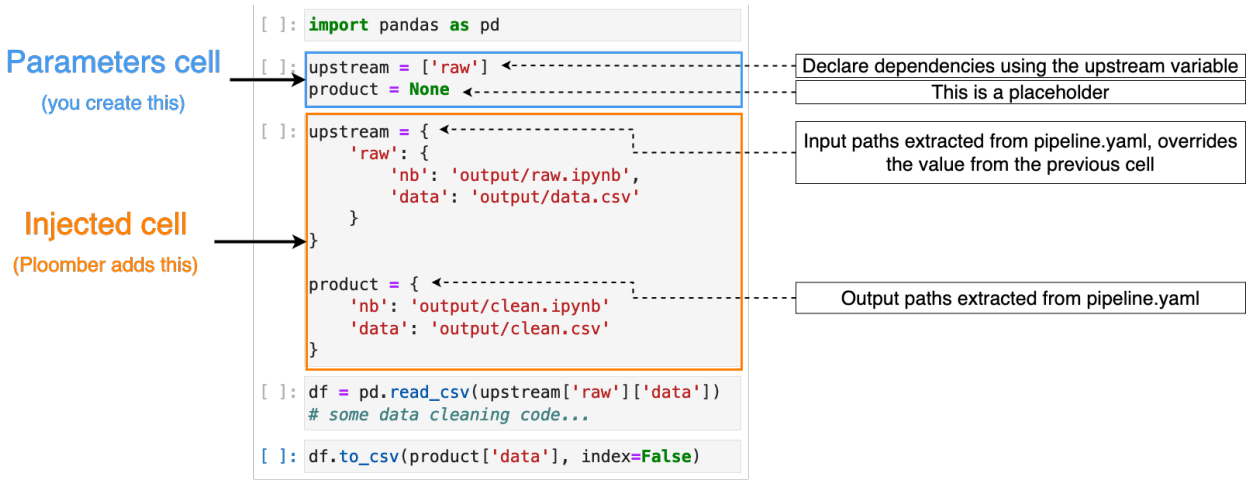
Note: For tips on troubleshooting pipeline loading, see *Troubleshooting pipeline loading*.

Let's review the contents of a sample `clean.py` file:

```
import        as

# %% tags=["parameters"]
         =  'raw'
        = None

# %%
   =   .                  'raw'   'data'
# some data cleaning code...

# %%
# store clean data
  .                   'data'         =False
```

This code will break if we run it: We declared `raw` as an upstream dependency, but we don't know where to load our inputs from, or where to save our outputs.

When executing your pipeline, Ploomber injects a new cell into each script/notebooks, with new `product` and `upstream` variables that replace the original ones by extracting information from the `pipeline.yaml`:



As you can see in the image, the task in the picture has an upstream dependency called `raw`. Thus, the injected cell has a dictionary that gives you the products of `raw`, which we use as input, and a new `product` variable that we use to store our outputs.

The cell injection process also happens when opening the notebook/script in Jupyter. Learn more about *Cell injection and Jupyter integration*.

Note: When using `jupyter notebook`, scripts open automatically as notebooks. If using `jupyter lab`, you have to right-click and select the notebook option.

Since scripts/notebooks always create an executed notebook, you must specify where to save such a file. A typical task declaration looks like this:

```
tasks
    source  plot.py
    # output notebook
    product  output/plots.ipynb
```

If the source script/notebook generates more than one output, create a dictionary under `product`:

```
tasks
    source  plot.py
    product
      # if the script generates other products, use "nb" for the notebok
      nb  output/plots.ipynb
      # ...and any other keys for other files
      data  output/data.csv
```

Note: The name of keys in the product dictionary can be chosen freely so as to be descriptive of the outputs (e.g. `data`, `data_clean`, `model`, etc.)

To inject cells manually, users can run:

```
ploomber nb --inject
```

However, if the same source appears more than once, `--inject` will pick the first declared task and inject those parameters.

Here is an example where `template.ipynb` appears in two different tasks

```
tasks
  source  template.ipynb
   name  task-a
   product  output/template-task-a.ipynb
   params
   some_param  param-a

  source  template.ipynb
   name  task-a-suffix
   product  output/template-task-a-suffix.ipynb
   params
   some_param  param-a-suffix

  source  template.ipynb
   name  task-b-suffix
   product  output/template-task-b-suffix.ipynb
   params
   some_param  param-b-suffix
```

By using the `inject-priority` parameter in `setup.cfg`, we can specify which set of parameters to inject:

To inject `param-a` to `task-a` :

```
[ploomber]
entry-point = path/to/pipeline.yaml
inject-priority = task-a
```

Use wildcards to inject multiple parameters (*)

To inject `param-a-suffix` to `task-a-suffix`, and `param-b-suffix` to `task-b-suffix` :

```
[ploomber]
entry-point = path/to/pipeline.yaml
inject-priority = *-suffix
```

This covers scripts and notebooks as tasks, if you want to learn how to use functions as tasks, keep scrolling, otherwise, *skip to the end.*

It is also possible to use placeholders in the `pipeline.yaml` file. For example, the following file uses a placeholder `some_param`.

```
# Content of pipeline.yaml
tasks
  source  print.py
   name  print
   product
      nb  'output/{{some_param}}/notebook.html'
   papermill_params
      log_output  True
   params
      some_param  '{{some_param}}'
```

This placeholder should be defined inside the `env.yaml` file, more on that here.

Tasks: functions

You can also use functions as tasks, the following section explains how.

`upstream` and `product`

The only requirement for a function to be a valid task is to have a `product` parameter.

```
import        as

def clean
   # save output using the product argument
     .
```

Note: If the function generates many products, this becomes a dictionary, for example: `product['one']`, and `product['another']`.

If the task has upstream dependencies, add an `upstream` parameter:

```
import        as

def clean
            =   .                 'task_name'
     .
```

When resolving dependencies, Ploomber will look for references such as `upstream['task_name']`, then, during execution, Ploomber will pass the requested inputs. For example, `upstream={'task_name': 'path/to/product/from/upstream.csv'}`

This covers scripts and functions as tasks, if you want to learn how to use SQL scripts as tasks, keep scrolling, otherwise, *skip to the end.*

Tasks: SQL

SQL tasks require more setup because you have to configure a `client` to connect to the database. We explain the `product` and `upstream` mechanism here; an *upcoming guide* describes how to configure database clients.

`upstream` and `product`

SQL scripts require placeholders for `product` and `upstream`. A script that has no upstream dependencies looks like this:

```
CREATE TABLE             AS -- {{product}} is a placeholder
SELECT * FROM          WHERE           > 10
```

In your `pipeline.yaml` file, specify `product` with a list of 3 or 2 elements: `[schema, name, table]` or `[name, table]`. If using a view, use `[schema, name, view]`. For example:

Say you have `product: [schema, name, table]` in your `pipeline.yaml` file. The `{{product}}` placeholder is replaced by `schema.name`:

```
CREATE TABLE schema.name AS
SELECT * FROM          WHERE           > 10
```

If the script has upstream dependencies, use the `{{upstream['task_name']}}` placeholder:

```
CREATE TABLE             AS
SELECT * FROM            'task_name'    WHERE           > 10
```

`{{upstream['task_name']}}` tells Ploomber to run the task with the name `'task_name'` and to replace `{{upstream['task_name']}}` with the product of such task.

Clients

To establish a connection with a database, you have to configure a `client`. All databases that have a Python driver are supported, including systems like Snowflake or Apache Hive. To learn more, see the *SQL guide*.

Where to go from here

We've created many **runnable templates** to help you get up and running, check out our *Downloading templates* guide.

If you want to read about **advanced features**, check out *User Guide*.

The `pipeline.yaml` API offers a concise way to declare pipelines, but if you want complete flexibility, **you can use the underlying Python API**, *Click here to learn more*, or click here to see an example.

To run this locally, install Ploomber and execute: `ploomber examples -n guides/intro-to-ploomber`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.1.5 Intro to Ploomber

Your first Python pipeline

Introductory tutorial to learn the basics of Ploomber.

Ploomber Tutorial Intro

We'll forcast the relation between testing and active covid-19 cases.

We'll see today how you can improve your work:

- Run 100s of notebooks in parallel
- Parameterize your workflows
- Easily generate HTML/PDF reports

For a deeper dive, try the first-pipeline guide or the basic concepts overview. If YAML, Jupyter and notebooks sounds like a distant cousin, please check our basic concepts guide.

Parallelization

- Ploomber creates a pipeline for you, so you can run independent tasks simultaneously.
- It also cache the results so you don't have to wait. You can drop the `force=True` (last line) and rerun this cell.

In here we'll train 4 different models simultaneously, and see it in a graph:

[1]:
```
from          import
from               import
from                  import
from                   import

from               import
     =         './pipeline.yaml'
    =     .
# dag.executor = Parallel()
      =    .           =True
```

```
fatal: ref HEAD is not a symbolic ref
```

```
  0%|          | 0/7 [00:00<?, ?it/s]
```

```
Executing:   0%|          | 0/6 [00:00<?, ?cell/s]
```

```
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
```

```
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
```

```
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
```

```
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
```

```
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
```

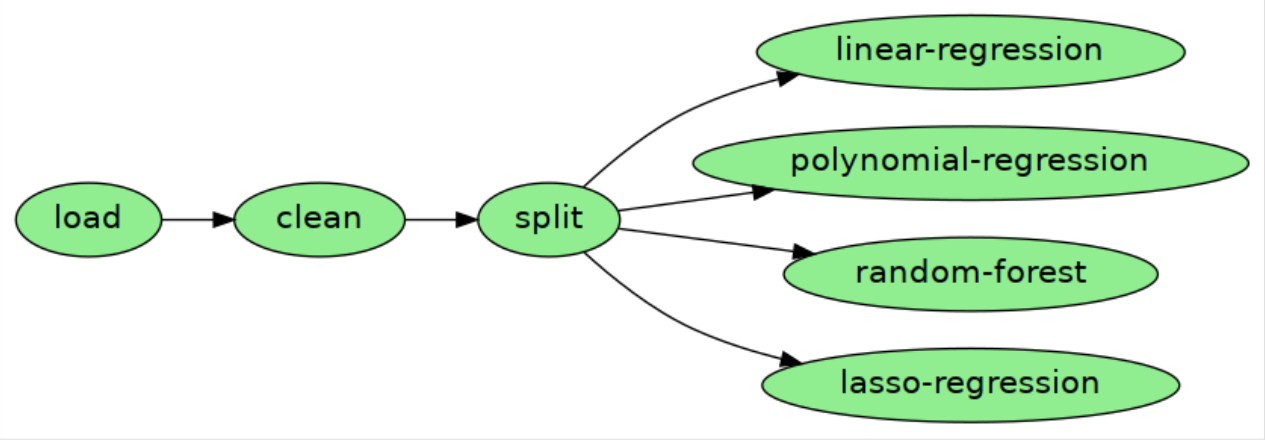
```
Executing:   0%|          | 0/8 [00:00<?, ?cell/s]
```

```
/home/prem/Documents/projects/ploomber/ploomberw/ploomber/src/ploomber/executors/serial.
↪py:149: UserWarning:
============================ DAG build with warnings ============================
- NotebookRunner: linear-regression -> MetaProduct({'nb': File('output/...ession.ipynb')}
↪) -
- /home/prem/Documents/projects/ploomber/ploomber-projectsw/ploomber-projects/guides/
↪intro-to-ploomber/tasks/linear-regression.py -
Output '/home/prem/Documents/projects/ploomber/ploomber-projectsw/ploomber-projects/
↪guides/intro-to-ploomber/output/linear-regression.ipynb' is a notebook file. nbconvert_
↪export_kwargs {'exclude_input': True} will be ignored since they only apply when␣
↪exporting the notebook to other formats such as html. You may change the extension to␣
↪apply the conversion parameters
============================== Summary (1 task) ==============================
NotebookRunner: linear-regression -> MetaProduct({'nb': File('output/...ession.ipynb')})
============================ DAG build with warnings ============================

  warnings.warn(str(warnings_all))
```

[2]:
```
   .
```

```
  0%|          | 0/7 [00:00<?, ?it/s]
```

[2]:



Parameterize workflows

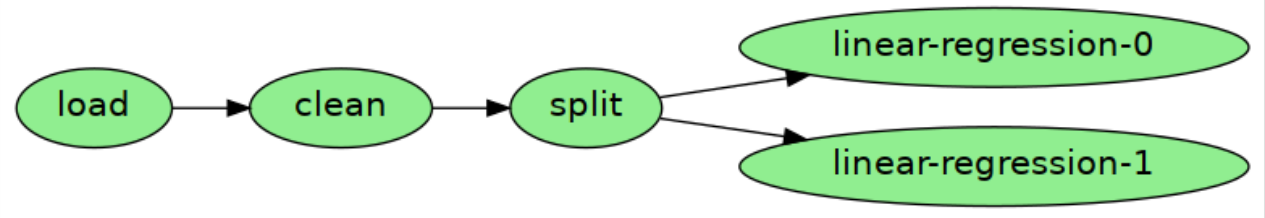
- In many cases, you'd run your analysis with different parameters/different data slices
- Ploomber allows you to parametrize workflows easily
- Here we're training a linear regression with different parameters, using a notebook as template

[3]:

```
from                import
     =          './pipeline-params.yaml'
    =      .
      =     .            =True


   .
```

```
fatal: ref HEAD is not a symbolic ref
  0%|          | 0/5 [00:00<?, ?it/s]
Executing:   0%|          | 0/6 [00:00<?, ?cell/s]
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
Executing:   0%|          | 0/7 [00:00<?, ?cell/s]
  0%|          | 0/5 [00:00<?, ?it/s]
```

[3]:



Caching optimization

Note that the previous table has load ran as fail?

This task ran in a previous pipeline so there's no point of reruning it. (we can force it to run if needed).

In the next table, all of the pipeline results were cached so we can focus on code that changed only, saving hours of compute time.

[4]:
```
      =    .
```

```
0it [00:00, ?it/s]
```

[4]:
```
name                Ran?      Elapsed (s)    Percentage
------------------  ------  -------------  ------------
load                False               0             0
clean               False               0             0
split               False               0             0
linear-regression-0 False               0             0
linear-regression-1 False               0             0
```

Automated reports

In case we have a dataset to track/a stakeholder report, we can generate it as part of our workflow. We created the report as part of our first cell pipeline build, so we can consume it immediately. Let's load our stakeholder report from our previous linear regression task:

[5]:
```
# open each specific html report/data if exist
from                 import
from         import

       = "./output/linear-regression.html"
if             .
                    =            ='100%'        ='500px'
else
          "Report doesn't exist - please run the notebook sequentially"
```

```
Report doesn't exist - please run the notebook sequentially
```

Interactive reporting

Compare your previous experiments interactively

[6]:
```
from                   import
# ids to identify each experiment
    =
      'linear-regression'  'polynomial-regression'  'random-forest'  'lasso-regression'


# output files
      =  f'output/{ }.ipynb' for   in
```

(continues on next page)

(continued from previous page)

```
    =                          =          =

    'plot'
```

[6]:
```
<sklearn_evaluation.nb.NotebookCollection.HTMLMapping at 0x7ffa0bfaf040>
```

Where to go from here

Use cases

- Machine Learning
- Research Projects
- Analytics
- SQL Pipelines

Community support

Have questions? Ask us anything on Slack.

Resources

Bring your own code! Check out the tutorial to migrate your code to Ploomber.

Want to dig deeper into Ploomber's core concepts? Check out the basic concepts tutorial.

Want to start a new project quickly? Check out how to get examples.

2.2 Use Cases

2.2.1 Machine Learning

Ploomber has many features specifically tailored to accelerate Machine Learning workflows.

Tip: Check out our sklearn-evaluation library. It contains a large collection of Machine Learning evaluation plots, an experiment tracker, and many other features!

Data cleaning and feature engineering

Data cleaning and feature engineering are highly iterative processes, Ploomber accelerates them via *incremental builds*, which allow you to introduce changes to your pipeline and bring results up-to-date without having to re-compute everything from scratch.

Experiment tracking

Ploomber also plays nicely with experiment trackers, allowing you to train hundreds of models and track the results.

Instructions

```
pip install ploomber
ploomber examples -n templates/mlflow -o ploomber-mlflow
```

Parallel experiments

To help you find the best performing model, Ploomber allows you to parallelize Machine Learning experiments.

```
pip install ploomber
ploomber examples -n cookbook/grid -o grid
```

```
pip install ploomber
ploomber examples -n cookbook/nested-cv -o nested-cv
```

Large-scale model training

If one machine isn't enough, you can parallelize training jobs in a cluster by *exporting your pipeline* to any of our supported platforms (Kubernetes, Airflow, and AWS Batch).

Deployment

Once you find the best performing model, you can deploy it for *batch processing* or as an *online API*.

2.2.2 Research Projects

Ploomber can help you manage your research project to enhance reproducibility and to run more experiments faster. Click here to see a sample project.

Faster iterations

Thanks to *incremental builds*, you can make small changes to your data analysis code and quickly bring your results up-to-date, since Ploomber will only execute the code that has changed since your last run.

Run (and organize) more experiments

Ploomber allows you to run many experiments in parallel. You can *parametrize pipelines* to run the same code with different configurations.

Furthermore, you can quickly generate all the parameter combinations from a *grid*. If one machine isn't enough, *export to systems* like Kubernetes or SLURM easily.

```
pip install ploomber
ploomber examples -n cookbook/grid -o grid
```

Ensure reproducibility

Since Ploomber generates an output notebook (that may include any number of tables or charts) whenever you execute your pipeline, you can easily bookkeep the results of each experiment. Whenever you make changes, such executed notebooks from previous runs can help you verify the reproducibility of your results.

Share your analysis

Ploomber can orchestrate all your data analysis for you if you need someone else to run your code, all they have to do is execute the following command:

```
ploomber build
```

2.2.3 Analytics

Ploomber is a fantastic tool for data manipulation and generating analytical reports.

```
pip install ploomber
ploomber examples -n templates/google-cloud -o google-cloud
```

```
pip install ploomber
ploomber examples -n templates/exploratory-analysis -o exploratory-analysis
```

Modularize your project

Instead of coding everything in a single notebook (which is difficult to maintain and collaborate), you can quickly break down your analysis into multiple parts.

Faster iterations

Finding data insights is an iterative process, with Ploomber's *incremental builds* you can rapidly iterate on your data since the framework skips redundant computations and only executes tasks whose source code has changed since the last execution.

Automated report generation

Once your pipeline is ready, you can easily create HTML reports from your scripts/notebooks. Just change the extension of the task, and Ploomber will automatically convert the output for you.

```
tasks
    source  tasks/plot.py
    # execute your .py file and generate an .html version of it
    # all tables and charts are included
    product  output/report.html
```

2.2.4 SQL Pipelines

Ploomber comes with built-in support for SQL. You provide SQL scripts and Ploomber manages connections to the database and orchestrates execution for you.

Tip: Check out our JupySQL library. It allows you to run SQL in a Jupyter notebook: `result = %sql SELECT * FROM table`

Process with SQL and Python

With data warehouses such as Snowflake, using SQL for transforming data can significantly simplify the development process since the warehouse takes care of scaling your code.

You can use Ploomber and SQL to process large datasets quickly, then download the data to continue your analysis with Python for plotting or training a Machine Learning model

```
pip install ploomber
ploomber examples -n templates/google-cloud -o google-cloud
```

```
pip install ploomber
ploomber examples -n templates/spec-api-sql -o spec-api-sql
```

Uploading batch predictions to a database

If you're working on a Machine Learning whose predictions must be uploaded to a database table, you can implement this with Ploomber.

ETL

Ploomber allows you to write ETL SQL pipelines.

```
pip install ploomber
ploomber examples -n templates/etl -o etl
```

2.3 User Guide

2.3.1 Downloading templates

Use our pre-configured templates as a starting point for your projects.

Selected templates

Tip: Click on the template link to see the source code on GitHub. Once there, you'll see an option to launch a free, hosted JupyterLab.

- **Exploratory Data Analysis**
    1. *Basic EDA example*: Load and clean data. Then create HTML reports with visualizations. (templates/exploratory-analysis)
- **Machine Learning**
    1. *Basic ML example*: Get data, clean it, and train a model. (templates/ml-basic)
    2. *Intermediate ML example*: Create training and batch serving pipelines. (templates/ml-intermediate)
    3. *Online API*: Deploy pipeline as an API using Flask. (templates/ml-online)
    4. *Experiment grid + Mlflow*: Create a grid of experiments and track them with MLflow. (templates/mlflow)
- **SQL databases**
    1. *Basic SQL example*: Process data, dump it, and visualize with Python. (templates/spec-api-sql)
    2. *ETL*: dump data from remote storage, upload it to a database, process it, and visualize it with Python. (templates/etl)

Downloading a template

To download any of the examples:

```
ploomber examples -n {template} -o {output-directory}
```

For example, if you want to copy the *Basic EDA example* to the `eda` directory in your computer:

```
ploomber examples -n templates/exploratory-analysis -o eda
```

Tip: Once you download an example, you can explore it with Jupyter. Check out the *Jupyter integration* guide to learn more.

Once the download finishes, you'll need to install dependencies; you can use the `ploomber install` command. You may call `conda` or `pip` directly.

Listing all templates

To list all the available examples:

```
ploomber examples
```

Note that the command above will display three sections:

1. *Templates.* Pre-configured projects that you can use as a starting point.
2. *Cookbook.* Short examples to get something done quickly.
3. *Guides.* In-depth tutorials covering features in detail.

Note that both `Cookbook` and `Guides` are part of the documentation itself, and you can navigate to any of them using the left sidebar or download them to run them locally.

Templates structure

All templates follow the same structure:

1. `README.md`: Instructions to run the template.
2. `README.ipynb`: Same as `README.md` but in notebook format and with command outputs.
3. `environment.yml`: `conda` dependencies file.
4. `requirements.txt`: `pip` dependencies file.

Most templates contain a `pipeline.yaml` file, so you can run `ploomber build` to execute the pipeline, but there are a few exceptions. Check out the template's `README.md` for specifics.

Starting projects from scratch

If no template suits your needs, use the `ploomber scaffold` command to create a clean slate project. *Click here to learn how to scaffold projects*.

`ploomber scaffold` also comes with utilities to modify existing pipelines, so can use it to change any of the templates.

2.3.2 Command-line interface

Note: This is an introductory tutorial to the command line interface; for a complete API description, see: *Command line interface*.

Entry points

By default, the CLI looks for an `pipeline.yaml` file in certain standard locations (*Default locations*). If your pipeline exists in a non-standard location, pass the `--entry-point` argument.

The `pipeline.yaml` file is known as "entry point". However, this is not the only type of entry point (See this guide to learn more: *Spec API vs. Python API*).

Basic commands

Build pipeline (skips up-to-date tasks):

```
ploomber build
```

Forced build (runs all tasks, regardless of status):

```
ploomber build --force
```

Generate pipeline plot:

```
ploomber plot
```

New in Ploomber 0.18.2: You can plot the pipeline without installing extra dependencies. `pygraphviz` is still supported but optional. To learn more, *see this*.

Interactive sessions

Interactive sessions allow you to access the structure of your pipeline to help you test and debug:

```
ploomber interact
```

The command above starts a Python session, parses your pipeline, and exposes a `dag` variable (an instance of the *ploomber.DAG* class).

For example, to generate the plot:

```
   .
```

Get task names:

```

```

You can also interact with specific tasks:

```
     =     'task_name'
```

Tip: If using IPython or Jupyter, press `Tab` to get autocompletion when typing the task name: `dag['some_task']`

Get task's product:

```
    'some_task' .
```

If the product is a dictionary:

```
    'some_task' .        'product_name'
```

You can use this to avoid hardcoding paths to load products:

```
import        as

   =    .             'some_task' .
```

If you are working with Python tasks (functions, scripts, or notebooks), you can start a line by line debugging session:

```
    'some_task' .
```

Enter `quit` to exit the debugging session. Refer to The Python Debugger documentation for details.

To print the source code of a given task:

```
    'some_task' .
```

To find the source code location of a given task:

```
    'some_task' .       .
```

Get upstream dependencies:

```
    'some_task' .
```

Get downstream tasks:

```
   .                'some_task'
```

Other commands

Some commands didn't cover here:

- `examples`: *Download examples*
- `install`: Install dependencies
- `nb` (short for notebook): Manage notebooks and scripts
- `report`: Generate a pipeline report
- `scaffold`: *Create a new project*
- `status`: Pipeline status summary
- `task`: Execute a single task

See the CLI API documentation *Command line interface* for a detailed overview of each command.

Enabling Completion

To configure autocompletion for the CLI, you need to configure your shell.

If using **bash**, add this to `~/bashrc`:

```
    "$(                    =zsh_source ploomber)"
```

If using **zsh**, add this to `~/.zshrc`:

```
    "$(                    =zsh_source ploomber)"
```

If using **fish**, add this to `~/.config/fish/completions/ploomber.fish`:

```
    (env                     =fish_source ploomber)
```

2.3.3 Jupyter integration

Note: This guide is applicable if running JupyterLab >=2.x. If running older versions or using other editors (such as VSCode or PyCharm), check out the *Other editors (VSCode, PyCharm, etc.)* guide.

Ploomber integrates with Jupyter to make it easy to create multi-stage pipelines composed of small notebooks. Breaking down logic in multiple steps allows you to develop modularized pipelines that are easier to maintain and deploy.

Before executing scripts or notebooks, Ploomber injects a new cell that replaces the `upstream` variable at the top of the notebook (which only contains dependency names) with a dictionary that maps these names to their corresponding output files to use as inputs in the current task.

For example if a Python script (`task.py`) declares the following dependency:

```
        =  'another-task'
```

And `another-task` has the following `product` definition:

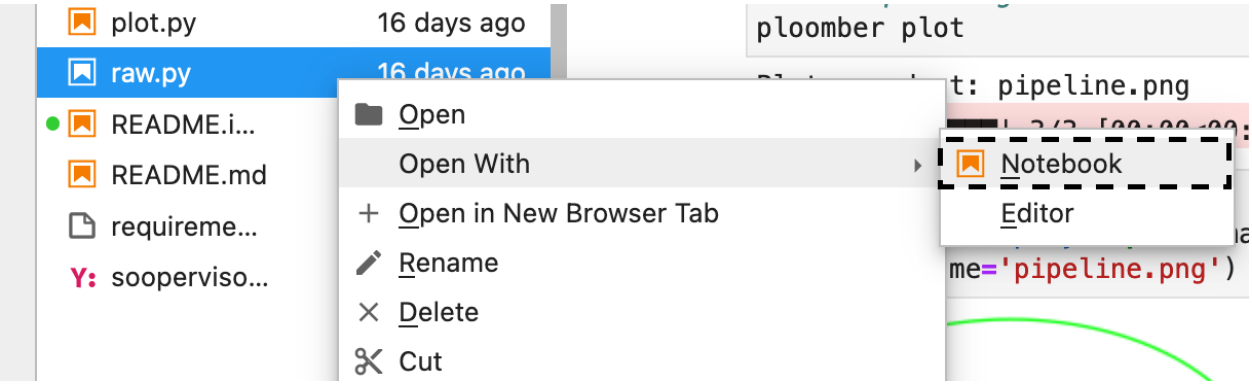
```
tasks
      source  another-task.py
      product
        nb  output/another-task.ipynb
        data  output/another-task.parquet
```

The following cell will be injected in `task.py` before execution:

```
# this is injected automatically
        =  'another_task'   'nb'  'output/another-task.ipynb'
                            'data'  'output/another-task.parquet'
```

The cell injection process happens during execution and development, allowing you to develop pipelines interactively.

Note: When using `jupyter notebook`, scripts automatically render as notebooks. If using `jupyter lab`: `Right-click` -> `Open With` -> `Notebook` as depicted below:



Note: If you want to configure JuptyerLab to open `.py` files as notebooks with a single click, see the *corresponding section*.

Important: *Task-level* and *DAG-level* hooks are **not** executed when opening scripts/notebooks in Jupyter.

Interactive development

You can develop entire pipelines without leaving Jupyter. The fastest way to get started is to use the `ploomber scaffold` command, which creates a base project, check out the guide to learn more: *Scaffolding projects*.

Once you have a `pipeline.yaml` file, you may add new tasks and run `ploomber scaffold` again to create base scripts. For example, say you create a `pipeline.yaml` like this:

```
tasks
    source  scripts/get.py
    product
      nb  output/get.ipynb
      data  output/get.csv

    source  scripts/clean.py
    product
      nb  output/clean.ipynb
      data  output/clean.csv

    source  scripts/fit.py
    product
      nb  output/fit.ipynb
      model  output/model.pickle
```

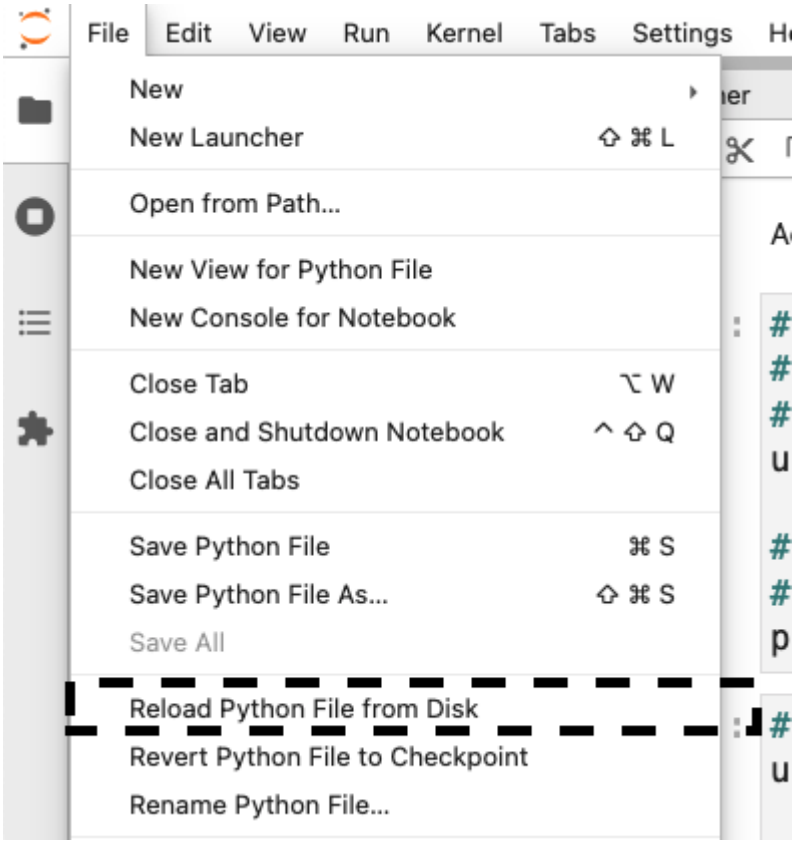
Once you execute `ploomber scaffold`, you'll see the three new scripts under the `scripts/` directory. You can then start adding the relationships between tasks.

The `upstream` variable

Let's say your `scripts/clean.py` script cleans some raw data. That means you want to use the raw data as input (which is downloaded by `scripts/get.py`), you can modify the `upstream` variable to establish this execution dependency.

```
# ensure we get the data, and then we clean it
        =  'get'
```

To inject the cell, reload the file from disk:



Then, you'll see something like this:

```
# injected cell
        = 'get' 'nb' 'output/clean.ipynb' 'data' 'output/clean.csv'
```

Now you can continue developing your cleaning logic without hardcoding any paths. Furthermore, when executing your pipeline, Ploomber will run `scripts/get.py` and then `scripts/clean.py`

Important: Ploomber needs to parse your `pipeline.yaml` file to inject cells in your scripts/notebooks; if an error happens during the parsing process, you won't see any injected cells. Check out the *Troubleshooting* section below for details.

Choosing the source format

Ploomber supports scripts and notebooks as source formats for tasks. We recommend using `.py` files, but you can use the traditional `.ipynb` format if you prefer so. As long as your file has a tag named `parameters`, it will work fine (click here to learn how to add the `parameters` cell)

The advantage of using `.py` files is that they're much easier to manage with git, the disadvantage is that `.py` only contain code (not output), so after editing your `.py` file, you need to run the task to create the executed notebook (the one you declare as a product of the task).

However, if you want a more ipynb-like experience with `.py` files, you can use jupytext's pairing feature to sync the output of a `.py` to a `.ipynb` file.

We rely on Jupytext for the `.py` to `.ipynb` conversion so that you can use any of the `.py` flavors, here are some examples:

Light format

```
# + tags=["parameters"]
          = None
         = None

# +
# another cell
```

Percent format

```
# %% tags=["parameters"]
          = None
         = None

# %%
# another cell
```

Check out Jupytext documentation for more details on the supported formats.

Activating the Jupyter extension

Note: For tips on troubleshooting pipeline loading, see *Troubleshooting pipeline loading*.

In most cases, the extension configures when you install Ploomber; you can verify this by running:

```
jupyter serverextension list
```

If Ploomber appears in the list, it means it's activated. If it doesn't show up, you can manually activate it with:

```
jupyter serverextension enable ploomber
```

To disable it:

```
jupyter serverextension disable ploomber
```

Important: If you want to use the extension in a hosted environment (JupyterHub, Domino, SageMaker, etc.), ensure Ploomber is installed **before** JupyterLab spins up. Usually, hosted platforms allow you to write a custom start script: add a `pip install ploomber` line, and you'll be ready to go. If you cannot get the extension to work, post a question in the `#ask-anything` channel on Slack. Alternatively, you may replicate the extension's functionality using the command line, check out the *this guide* to learn more.

Custom Jupyter pipeline loading

When you start the Jupyter app (via the `jupyter notebook/lab` command), the extension looks for a `pipeline.yaml` file in the current directory and parent directories. If it finds one, it will load the pipeline and inject the appropriate cell if the existing file is a task in the loaded pipeline.

If your pipeline spec has a different name, you can create a `setup.cfg` file and indicate what file you want to load. Note that **changing the default affects both the command-line interface and the Jupyter plug-in**.

```
[ploomber]
entry-point = path/to/pipeline.yaml
```

Note that paths are relative to the parent directory of `setup.cfg`.

Alternatively, you can set the `ENTRY_POINT` environment variable. For example, to load a `pipeline.serve.yaml`:

```
export ENTRY_POINT=pipeline.serve.yaml
jupyter lab
```

Important: `export ENTRY_POINT` must be executed in the same process that spins up JupyterLab. If you change it, you'll need to start JupyterLab again

---

Note that `ENTRY_POINT` must be a file name and not a path. When you start Jupyter, Ploomber will look for that file in the current and parent directories until it finds one.

New in version 0.19.6: Support for switching entry point with a `setup.cfg` file

Troubleshooting pipeline loading

Note: For tips on activating the Jupyter extension, see *Activating the Jupyter extension*.

If a pipeline is not detected, the Jupyter notebook application will work as expected, but no cell injection will happen. You can see if Ploomber could not detect a pipeline by looking at the messages displayed after initializing Jupyter (the terminal window where you executed the `jupyter notebook/lab` command, you'll see something like this:

```
[Ploomber] Skipping DAG initialization since there isn't a project root in the current␣
↪or parent directories. Error message: {SOME_MESSAGE}
```

The message above means that Ploomber could not locate a `pipeline.yaml` file to use for cell injection, take a look at the entire error message as it will contain more details to help you fix the problem. A common mistake is not to include a `pipeline.yaml` file in the same directory (or parent) of the script/notebook you're editing.

If a `pipeline.yaml` is found but fails to initialize, the Jupyter console will show another error message:

```
[Ploomber] An error occurred when trying to initialize the pipeline.
```

A common reason for this is an invalid `pipeline.yaml` file.

Note that even if your pipeline is missing or fails to initialize, Jupyter will start anyway, so ensure to take a look at the console if you experience problems.

Another common situation is `ModuleNotFoundError` errors. Jupyter must parse your pipeline in the process that runs the Jupyter application itself. If your pipeline contains dotted paths (e.g., tasks that are Python functions, task hooks,

task clients, etc.), loading the pipeline will fail if such dotted paths are not importable. Scripts and notebooks are handled differently. Hence, a pipeline whose tasks are all notebooks/scripts won't have this issue.

If you cannot find the problem, you can move to a directory that stores any of the scripts that aren't having the cell injected, start a Python session and run:

```
from          import
```

`lazily_load_entry_point` is the function that Ploomber uses internally to initialize your pipeline. Calling this function allows you to replicate the same conditions when initializing your pipeline for cell injection.

Detecting changes

Ploomber parses your pipeline whenever you open a file to detect changes. The parsing runtime depends on the number of tasks, and although it is fast, it may slow down file loading in pipelines with lots of tasks. You can turn off continuous parsing by setting `jupyter_hot_reload` (in the `meta` section) option to `False`. You'll have to restart Jupyter if you turn this option off to detect changes.

Managing multiple pipelines

Jupyter can detect more than one pipeline in a single project. There are two ways to achieve this.

The first one is to create sibling folders, each one with its own `pipeline.yaml`:

```
some-pipeline/
    pipeline.yaml
    some-script.py
another-pipeline/
    pipeline.yaml
    another-script.py
```

Since Ploomber looks for a `pipeline.yaml` file in the current directory and parents, it will correctly find the appropriate file if you open `some-script.py` or `another-script.py` (assuming they're already declared as tasks in their corresponding `pipeline.yaml`.

Important: If using Python functions as tasks, you must use different module names for each pipeline. Otherwise, the module imports first will be cached and used for the other pipeline. See the following example.

```
some-pipeline/
    pipeline.yaml
    some_tasks.py
another-pipeline/
    pipeline.yaml
    other_tasks.py
```

The second option is to keep a unique project root and name each pipeline differently:

```
pipeline.yaml
some-script.py
pipeline.another.yaml
another-script.py
```

In this case, Ploomber will load `pipeline.yaml` by default, but you can switch this by setting the `ENTRY_POINT` variable to the other spec. (e.g., `pipeline.another.yaml`). Note that the environment variable must be a filename and not a path.

Exploratory Data Analysis

There are two ways to use Ploomber in Jupyter. The first one is by opening a task file in Jupyter (i.e., the source file is listed in your `pipeline.yaml` file.

Another way is to load your pipeline in Jupyter to interact with it. This second approach is best when you already have some tasks, and you want to explore their outputs to decide how to proceed with further analysis.

Say that you have a single task that loads the data:

```
tasks
      source  load.py
      product
        nb  output/load.ipynb
        data  output/data.csv
```

If you want to explore the raw data to decide how to organize downstream tasks (i.e., for data cleaning). You can create a new notebook with the following code:

```
from               import

    =        .       .
```

Note that this exploratory notebook **is not** part of your pipeline (i.e., it doesn't appear in the `tasks` section of your `pipeline.yaml`), it's an independent notebook that loads your pipeline declaration.

The `dag` variable is an object that contains your pipeline definition. If you want to load your raw data:

```
import        as

   =    .             'load' .
```

Using the `dag` object avoids hardcoded paths to keep notebooks clean.

There are other things you can do with the `dag` object. See the following guide for more examples: *Interactive sessions*.

As your pipeline grows, exploring it from Jupyter helps you decide what tasks to build next and understand dependencies among tasks.

If you want to take a quick look at your pipeline, you may use `ploomber interact` from a terminal to get the `dag` object.

Opening `.py` files as notebooks with a single click

It is now possible to open `.py` files as notebooks in `JuptyerLab` with a single click (this requires `jupytext>=1.13.2`).

If using `ploomber>=0.14.7`, you can enable this with the following command:

```
ploomber nb --single-click
```

To disable:

```
ploomber nb --single-click-disable
```

If running earlier versions of Ploomber, you can enable this by changing the default viewer for text notebooks. For instructions, see jupytext's documentation (click on the triangle right before the `With a click on the text file in JupyterLab` section).

2.3.4 Scaffolding projects

Note: This is a guide on `ploomber scaffold`. For API docs see *Create new project*.

You can quickly create new projects using the `scaffold` command:

```
ploomber scaffold
```

After running it, type a name for your project and press enter. The command will create a pre-configured project with a sample pipeline.

New in 0.16: `ploomber scaffold` now takes a positional argument. For example, `ploomber example my-project`.

By adding the `--empty` flag to scaffold, you can create a project with an empty `pipeline.yaml`:

```
ploomber scaffold --empty
```

Scaffolding tasks

Once you have a `pipeline.yaml` file, `ploomber scaffold` behaves differently, allowing you to create new task files quickly. For example, say you add the following task to your YAML file:

```
tasks
    # some existing tasks....

    # new task
      source  tasks/my-new-task.py
      product  output/my-new-task.ipynb
```

Executing:

```
ploomber scaffold
```

Will create a base task at `tasks/my-new-task.py`. This command works with Python scripts, functions, Jupyter notebooks, R Markdown files, R scripts, and SQL scripts.

`ploomber scaffold` works as long as your `pipeline.yaml` file is in a standard location (*Default locations*); hence, you can use it even if you didn't create your project with an initial call to `ploomber scaffold`.

By adding the `--entry-point/ -e`, you can specify a custom entry point. For example, if your spec is named `pipeline.serve.yaml`:

```
ploomber scaffold --entry-point pipeline.serve.yaml
```

Packaging projects

When working on larger projects, it's a good idea to configure them as a Python package. Packaged projects have more structure and require more configuration, but they allow you to organize your work better.

For example, if you have Python functions that you re-use in several files, you must modify your `PYTHONPATH` or `sys.path` to ensure that such functions are importable wherever you want to use them. If you package your project, this is no longer necessary since you can install your project using `pip`:

```
pip install --editable path/to/myproject
```

Installing with *pip* tells Python to treat your project as any other package, allowing you to import modules anywhere (in a Python session, notebook, or other modules inside your project).

You can scaffold a packaged project with:

```
ploomber scaffold --package
```

Note that the layout is different. At the root of your project, you'll see a `setup.py` file, which tells Python that this directory contains a package. The `pipeline.yaml` file is located at `src/{package-name}/pipeline.yaml`. All your pipeline's source code must be inside the `src/{package-name}` directory. Other files such as exploratory notebooks or documentation must be outside the `src` directory.

For example, say you have a `process_data` function defined at `src/my_awesome_package/processors.py`, you may start a Python session and run:

```
from                    import

          .
```

Such import statement works independently of the current working directory; you no longer have to modify the `PYTHONPATH` or `sys.path`. Everything under `src/{package-name}` is importable.

Managing development and production dependencies

`ploomber scaffold` generates two dependencies files:

- `pip`: `requirements.txt` (production) and `requirements.dev.txt` (development)
- `conda`: `environment.yml` (production) and `environment.dev.yml` (development)

While not required, separating development from production dependencies is highly recommended. During development, we usually need more dependencies than we do in production. A typical example is plotting libraries (e.g., matplotlib or seaborn); we need them for model evaluation but not for serving predictions. Fewer production dependencies make the project faster to install, but more importantly, it reduces dependency resolution errors. The more dependencies you have, the higher the chance of running into installation issues.

After executing `ploomber scaffold` command, and editing your dependency files, you can run:

```
ploomber install
```

To install dependencies. Furthermore, it configures your project if it's a package (i.e., you created it with `ploomber scaffold --package`).

During deployment, only install production dependencies and ignore development ones.

If you want to learn more about the `ploomber install` command, check out the CLI documentation: *install*.

If you want to know more about dependency management, check out this post in our blog.

Locking dependencies

Changes in your dependencies may break your project at any moment if you don't pin versions. For example, if you train a model using scikit-learn version 0.24 but only set *scikit-learn* as a dependency (without the version number). As soon as scikit-learn introduces breaking API changes, your project will fail. Therefore, it is essential to record specific versions to prevent broken projects.

You can do so with:

```
ploomber install
```

Such command detects whether to use pip/conda and creates lock files for development and production dependencies; lock files contain an exhaustive list of dependencies with a specific version.

2.3.5 Refactoring legacy notebooks

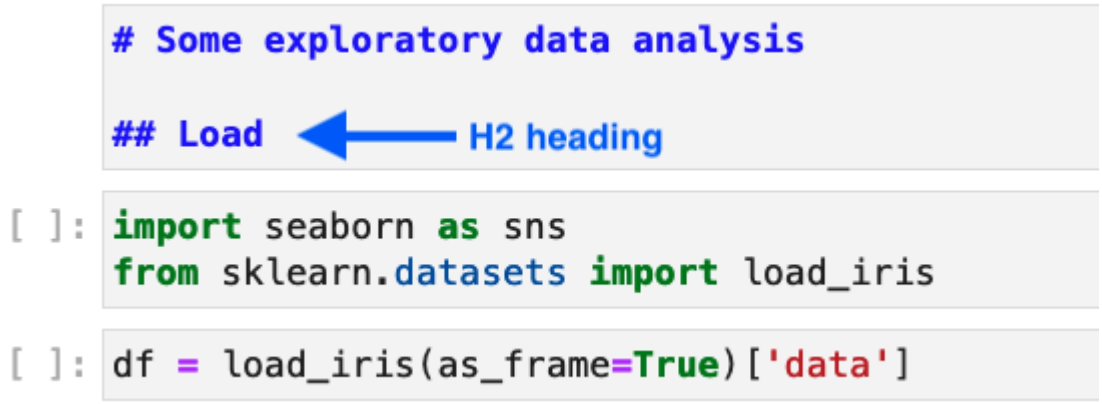
This tutorial shows how to convert legacy notebooks into Ploomber pipelines.

Note: If you don't have a sample notebook, download one from here.

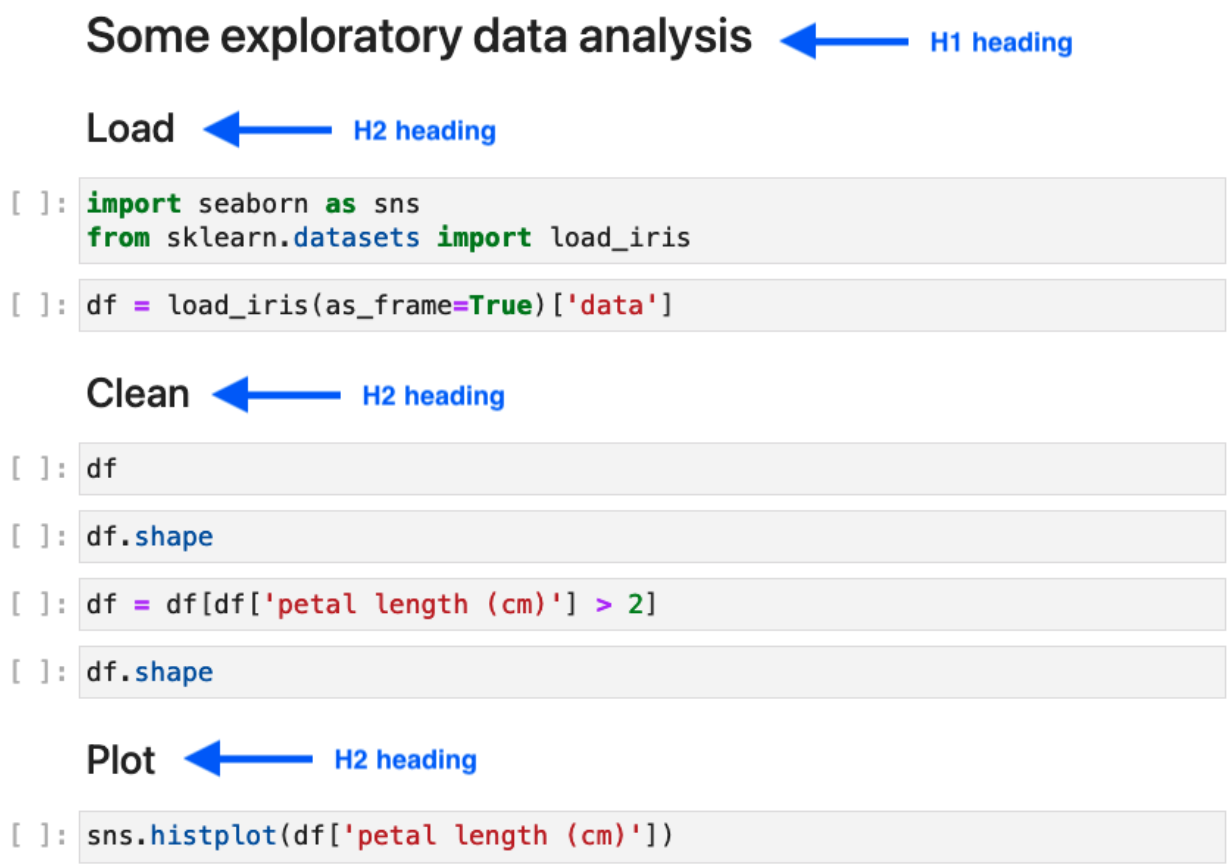
or execute:

```
curl -O https://raw.githubusercontent.com/ploomber/soorgeon/main/examples/machine-
↪learning/nb.ipynb
```

The only requirement for your notebook is to separate sections with H2 headings:



Here's an example notebook with three sections separated by H2 headings:



Once your notebook is ready, you can refactor it with:

```
# install soorgeon
pip install soorgeon

# refactor the nb.ipynb notebook
soorgeon refactor nb.ipynb
```

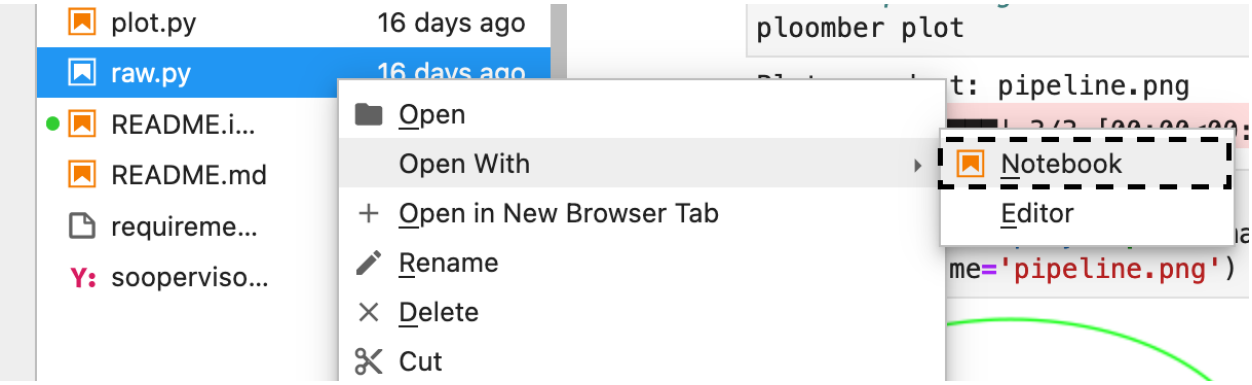
Tip: Sometimes, `soorgeon` may not be able to split your notebook sections, if so, run `soorgeon refactor nb.ipynb --single-task` to generate a pipeline with one task. If you have questions, send us a message on Slack.

The command above will generate a `pipeline.yaml` with your pipeline declaration and `.ipynb` tasks (one per section).

You can also tell Soorgeon to generate tasks in `.py` format:

```
# generate tasks in .py format (requires soorgeon>=0.0.13)
soorgeon refactor nb.ipynb --file-format py
```

Note that due to the *Jupyter integration*, **you can open .py files as notebooks in Jupyter**



To run the pipeline:

```
# install dependencies
pip install -r requirements.txt

# run Ploomber pipeline
ploomber build
```

That's it! Now that you have a Ploomber pipeline, you can benefit from all our features! If you want to learn more about the framework, check out the *basic concepts tutorial*.

Resources

- Soorgeon's user guide
- GitHub
- Interactive example
- Blog post series on notebook refactoring: Part I, and Part II

To run this locally, install Ploomber and execute: `ploomber examples -n guides/parametrized`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.3.6 Parametrized pipelines

Tutorial showing how to parametrize pipelines and change parameters from the command-line.

Often, pipelines perform the same operation over different subsets of the data. For example, say you are developing visualizations of economic data. You might want to generate the same charts for other countries.

One way to approach the problem is to have a for loop on each pipeline task to process all needed countries. But such an approach adds unnecessary complexity to our code; it's better to keep our logic simple (each task processes a single country) and take the iterative logic out of our pipeline.

Ploomber allows you to do so using parametrized pipelines. Let's see a sample using a `pipeline.yaml` file.

Spec API (`pipeline.yaml`)

```
# Content of pipeline.yaml
tasks
    source  print.py
    name  print
    product
      nb  'output/{{some_param}}/notebook.html'
    papermill_params
        log_output  True
    params
      some_param  '{{some_param}}'
```

The `pipeline.yaml` above has a placeholder called `some_param`. It is coming from a file called `env.yaml`:

```
# Content of env.yaml
some_param  default_value
```

When reading your `pipeline.yaml`, Ploomber looks for an `env.yaml` file. If found, all defined keys will be available to your pipeline definition. You can use these placeholders (placeholders are strings between double curly brackets) in any of the fields of your `pipeline.yaml` file.

In our case, we are using it in two places. First, we will save the executed notebook in a folder with the value of `some_param`; this will allow you to keep copies of the generated output in a different folder depending on your parameter. Second, if we want to use the parameter in our code, we have to pass it to our tasks; all tasks take an optional `params` with arbitrary parameters.

Let's see how the code looks like:

```
# Content of print.py
# + tags=["parameters"]
         = None
        = None
           = None

# +
      'some_param: '              ' type: '
```

Our task is a Python script, meaning that parameters are passed as an injected cell at runtime. Let's see what happens if we build our pipeline.

[1]:
```
%%capture captured
%%
               --      --
```

[2]:
```
def filter_output
    return       '\n'.
             for      in         .      .      '\n'
        if     .


                                  ='INFO:papermill:some_param'
```

```
INFO:papermill:some_param:  default_value  type:  <class 'str'>
```

We see that our param `some_param` is taking the default value (`default_value`) as defined in `env.yaml`. The command-line interface is aware of any parameters. You can see them using the `--help` option:

[3]:
```
%%sh
ploomber build --help
```

```
usage: ploomber [-h] [--log LOG] [--log-file LOG_FILE]
                [--entry-point ENTRY_POINT] [--force] [--skip-upstream]
                [--partially PARTIALLY] [--debug]
                [--env--some_param ENV__SOME_PARAM]

Build pipeline

optional arguments:
  -h, --help            show this help message and exit
  --log LOG, -l LOG     Enables logging to stdout at the specified level
  --log-file LOG_FILE, -F LOG_FILE
                        Enables logging to the given file
  --entry-point ENTRY_POINT, -e ENTRY_POINT
                        Entry point, defaults to pipeline.yaml
  --force, -f           Force execution by ignoring status
  --skip-upstream, -su  Skip building upstream dependencies. Only applicable
                        when using --partially
  --partially PARTIALLY, -p PARTIALLY
                        Build a pipeline partially until certain task
  --debug, -d           Drop a debugger session if an exception happens
  --env--some_param ENV__SOME_PARAM
                        Default: default_value
```

Apart from the default parameters from the `ploomber build` command, Ploomber automatically adds any parameters from `env.yaml`, we can easily override the default value. Let's do that:

[4]:
```
%%capture captured
%%
                --       --    --                          --
```

[5]:
```
                                  ='INFO:papermill:some_param'
```

```
INFO:papermill:some_param:  another_value  type:  <class 'str'>
```

We see that our task effectively changed the value!

Finally, let's see how the `output/` folder looks like:

[2]:
```
%%sh
tree output
```

```
output
├── another_value
│   └── notebook.html
└── default_value
    └── notebook.html

2 directories, 2 files
```

We have separate folders for each parameter, helping to keep things organized and taking the looping logic out of our pipeline.

Notes

- There are some built-in placeholders that you can use without having an `env.yaml` file. For example, `{{here}}` will expand to the `pipeline.yaml` parent directory. Check out the Spec API documentation for more information.
- This example uses a Python script as a task. In SQL pipeline, you can achieve the same effect by using the placeholder in the product's schema or a table/view name prefix.
- If the parameter takes many different values and you want to run your pipeline using all of them, calling them by hand might get tedious. So you have two options 1) write a bash script that calls the CLI with different value parameters or 2) Use the Python API (everything that the CLI can do, you can do with Python directly), take a look at the `DAGSpec` documentation.
- Parametrized `pipeline.yaml` files are a great way to simplify a task's logic but not overdo it. If you find yourself adding too many parameters, it's a better idea to use the Python AP directly; factory functions are the correct pattern for highly customized pipeline construction.
- Given that the two pipelines are entirely independent, we could even run them in parallel.

Python API (factory functions)

Parametrization is straightforward when using a factory function. If your factory takes parameters, they'll also be available in the command-line interface. Types are inferred from type hints. Let's see an example:

```
# Content of factory.py
from          import


def make                           = 10
        =
    # add tasks to your pipeline...
    return
```

Our function takes two parameters: `param` and `another`. Parameters with no default values (`param`) turn into positional arguments, and function parameters with default values convert to optional parameters (`another`). To see the same auto-generated API, you can use the `--help` command:

[7]:
```
%%sh
ploomber build --entry-point factory.make --help
```

```
usage: ploomber [-h] [--log LOG] [--log-file LOG_FILE]
                [--entry-point ENTRY_POINT] [--force] [--skip-upstream]
                [--partially PARTIALLY] [--debug] [--another ANOTHER]
                param

Build pipeline

positional arguments:
  param

optional arguments:
  -h, --help            show this help message and exit
  --log LOG, -l LOG     Enables logging to stdout at the specified level
  --log-file LOG_FILE, -F LOG_FILE
```

(continues on next page)

(continued from previous page)

```
                        Enables logging to the given file
  --entry-point ENTRY_POINT, -e ENTRY_POINT
                        Entry point, defaults to pipeline.yaml
  --force, -f           Force execution by ignoring status
  --skip-upstream, -su  Skip building upstream dependencies. Only applicable
                        when using --partially
  --partially PARTIALLY, -p PARTIALLY
                        Build a pipeline partially until certain task
  --debug, -d           Drop a debugger session if an exception happens
  --another ANOTHER
```

Note that the Python API requires more work than a `pipeline.yaml` file, but it is more flexible. [Click here] to see examples using the Python API.

2.3.7 Configuration (dev/prod)

In the previous guide (*Parametrized pipelines*), we saw how to use an `env.yaml` file to parametrize our pipeline and switch parameters from the command line.

Sometimes we want to change all the parameters at once. The most common scenario is to change configuration during development and production.

For example, say you're working on a Machine Learning pipeline whose `pipeline.yaml` looks like this:

```yaml
tasks

    source  get.py
    product
      nb  get.ipynb
      data  raw.csv
    params
      sample_pct  '{{sample_pct}}'

    source  get.py
    product
      nb  get.ipynb
      data  raw.csv

    source  get.py
    product
      nb  get.ipynb
      data  raw.csv
```

The pipeline above has one placeholder `'{{sample_pct}}'`, which controls which percentage of raw data to download. You may want to develop locally with a fraction of the data, say 20%, to iterate quickly. To smoke test quickly, you may run it with a smaller sample, say 1%. Finally, to train a model, you'll use 100% of the data.

Tip: You can use placeholders (e.g., `{{sample_pct}}`) anywhere in the `pipeline.yaml` file. Another typical use case is to switch the product location (e.g., `product: '{{product_directory}}/some-data.csv'`.

By default, Ploomber looks for an `env.yaml`. To enable rapid local development with 20% of the data, you may create an `env.yaml` file like this:

```
sample_pct  20
```

For smoke testing, `env.test.yaml`:

```
sample_pct  1
```

And for training, `env.train.yaml`:

```
sample_pct  100
```

To switch configurations, you can set the `PLOOMBER_ENV_FILENAME` environment variable to `env.test.yaml` in the testing environment and to `env.train.yaml` in the training environment.

Whenever `PLOOMBER_ENV_FILENAME` has a value, Ploomber uses it and looks for a file with such a name. Note that this must be a filename, not a path since Ploomber expects `env.yaml` files to exist in the same folder as the `pipeline.yaml` file. For example, if you're on Linux or macOS:

```
export PLOOMBER_ENV_FILENAME=env.train.yaml && ploomber build
```

Important: If you're using the Jupyter integration and want to see the changes reflected in the injected cell, you need to shut down Jupyter set `PLOOMBER_ENV_FILENAME`, and start Jupyter again.

Managing multiple pipelines

If your project has more than one pipeline, they'll likely need different `env.yaml` files.

Say you have two pipelines, one for training a model (`pipeline.yaml`) and one for serving it (`pipeline.serve.yaml`). You can create an `env.yaml` file to parametrize `pipeline.yaml` and an `env.serve.yaml` to parametrize `pipeline.serve.yaml`:

```
project/
    pipeline.yaml
    pipeline.serve.yaml
    env.yaml
    env.serve.yaml
```

The general rule is as follows: When loading a `pipeline.{name}.yaml`, extract the `{name}` portion. Then look for a `env.{name}.yaml` file, if such file doesn't exist, look for an `env.yaml` file. Note that the `PLOOMBER_ENV_FILENAME` environment variable overrides this process.

Alternatively, you may separate the pipelines into different directories, and put an `env.yaml` on each one:

```
project-a/
    pipeline.yaml
    env.yaml
project-b/
    pipeline.yaml
    env.yaml
```

`env.yaml` composition (DRY)

Note: New in version 0.18

In many cases, your development and production environment configuration share many values in common. To keep them simple, you may create an `env.yaml` (development configuration) and have your `env.prod.yaml` (production configuration) inherit from it:

```
key  value
key_another  dev-value
```

Then in your `env.prod.yaml`:

```
meta
  # import development config
  import_from  env.yaml

# no need to declare key: value here, it'll be imported from env.yaml

# overwrite value
key_another  production-value
```

Note that if the value in `import_from` is a relative path, it is considered so relative to the location of the env file (in our case `env.prod.yaml`).

You can switch values in `env.yaml` from the command line, to see how:

```
ploomber build --help
```

Example, if you have a `key` in your `env.yaml`:

```
ploomber build --env--key new-value
```

2.3.8 SQL Pipelines

This guide explains how to develop pipelines where some of all tasks are SQL scripts.

Note: This tutorial shows the built-in SQL features. However, this is not the only way for Ploomber to interact with databases. You may as well create functions (or scripts) that run queries in a database. The primary benefit of using the built-in features is that Ploomber manages many things for you (such as active connections, running queries in parallel, dumping tables to local files), so you only write `.sql` files.

Tip: Check out our JupySQL library. It allows you to run SQL in a Jupyter notebook: `result = %sql SELECT * FROM table`

Quick Start

If you want to take a look at the sample pipeline, you have a few options:

- Source code on Github

Or run it locally:

```
ploomber examples --name templates/spec-api-sql
```

You can also refer to this README file for more information on using SQL scripts to manipulate data in a database, dump a table, and plot it with Python.

Creating Sample Data

To create sample data, you can run the following code:

```
# create sample data
   setup
bash setup.sh
# move back to the original spec-api-sql folder
   ..
```

Connecting To Databases

Note: For a more detailed explanation on connecting to a database, see: *Database configuration*.

The first step to write a SQL pipeline is to tell Ploomber how to connect to the database, by providing a function that returns either a *ploomber.clients.SQLAlchemyClient* or a *ploomber.clients.DBAPIClient*. These two clients cover all databases supported by Python, even systems like Snowflake or Apache Hive.

`SQLAlchemyClient` takes a single argument, the database URI (Click here for documentation on sqlalchemy URIs.). As the name suggests, it uses SQLAlchemy under the hood, so any database supported by such library is supported as well. Below, there's is an example that connects to a local SQLite database:

```
from                  import

def get_client
    return                  'sqlite:///database.db'
```

If SQLAlchemy doesn't support your database, you must use *ploomber.clients.DBAPIClient* instead. Refer to the documentation for details.

Configuring The Task Client In `pipeline.yaml`

To configure your `pipeline.yaml` to run a SQL task, `source` must be a path to the SQL script. To indicate how to load the client, you have to include the `client` key:

```
tasks
    source  sql/create-table.sql
    client  clients.get_client
    # task declaration continues...
```

`client` must be a dotted path to a function that instantiates a client. If your `pipeline.yaml` and `clients.py` are in the same folder, you should be able to do this directly. If they are in a different folder, you'll have to ensure that the function is importable.

You can reuse the same dotted path in many tasks. However, since it is common for many tasks to query the same database, you may declare a task-level client like this:

```
clients
    # all SQLScript tasks use the same client instance
    SQLScript  config.get_client
    # all SQLDump tasks use the same client instance
    SQLDump  config.get_client

tasks
    source  sql/create-table.sql
    # no need to add client here
```

`SQLScript` (creates a table/view), and `SQLDump` (dump to a local file) are the two most common types of SQL tasks, let's review them in detail.

Creating SQL Tables/Views With `SQLScript`

If you want to organize your SQL processing in multiple steps, you can use `SQLScript` to generate one table/view per task. The declaration in the `pipeline.yaml` file looks like this:

```
tasks
    source  sql/create-table.sql
    client  clients.get_client
    product
```

`product` can be a list with three elements: `[schema, name, kind]`, or 2: `[name, kind]`. Where `kind` can be `table` or `view`.

A typical script (`sql/create-table.sql` in our case) looks like this:

```
DROP TABLE IF EXISTS

CREATE TABLE             AS
SELECT * FROM schema.
#          ...
```

This `DROP TABLE ... CREATE TABLE ..` format ensures that the table (or view) is deleted before creating a new version if the source code changes.

Note that we are using a `{{product}}` placeholder in our script, this will be replaced at runtime for the name value in `tasks[*].product` (in our case: `schema.name`.

`SQLScript` And Product's Metadata

Incremental builds (*What are incremental builds?*) allow you speed up pipeline execution. To enable this, Ploomber keeps track of source code changes. When tasks generate files (say `data.csv`), a metadata file is saved next to the product file (e.g., `.data.csv.metadata`).

To enable incremental builds in `SQLScript` tasks, you must configure a product metadata backend.

If you are using PostgreSQL, you can use *ploomber.products.PostgresRelation*; if using SQLite, you can use *ploomber.products.SQLiteRelation*. In both cases, metadata is saved in the same database where the tables/views are created. Hence, you can reuse the task client. Here's an example if using PostgreSQL:

```
meta
    # configure pipeline to use PostgresRelation by default
    product_default_class
        SQLScript  PostgresRelation

# same client for task and product
clients
    SQLScript  clients.get_pg_client
    PostgresRelation  clients.get_pg_client

tasks
    source  sql/create-table.sql
    product
```

For any other database, you have two options, either use *ploomber.products.SQLRelation* which is a product that does not save any metadata at all (this means you don't get incremental builds) or use *ploomber.products.GenericSQLRelation*, which stores metadata in a SQLite database.

A typical configuration to enable incremental builds looks like this:

```
meta
    product_default_class
        SQLScript  GenericSQLRelation

clients
    SQLScript  clients.get_db_client
    GenericSQLRelation  clients.get_metadata_client

tasks
    source  sql/create-table.sql
    name  some_task
```

Don't confuse the task's client with the product's client. **Task clients control where to execute the code. Product clients manage where to save metadata.**

Placeholders In SQL Scripts

You can reference the `product` list in your `pipeline.yaml` in your script using the `{{product}}` placeholder. For example `[schema, name, table]` renders to: `schema.name`.

To specify upstream dependencies, use the `{{upstream['some_task']}}` placeholder. Here's a complete example:

```
-- {{product}} gets replaced by the value in pipeline.yaml
DROP TABLE IF EXISTS

CREATE TABLE             AS
-- this task depends on the output generated by a task named "clean"
SELECT * FROM            'clean'
WHERE   > 10
```

Let's say our product is `[schema, name, table]` And the task named `clean` generates a product `schema.clean`, the script above renders to:

```
DROP TABLE IF EXISTS schema.name

CREATE TABLE schema.name AS
SELECT * FROM schema.
WHERE   > 10
```

If you want to see the rendered code for any task, execute the following in the terminal:

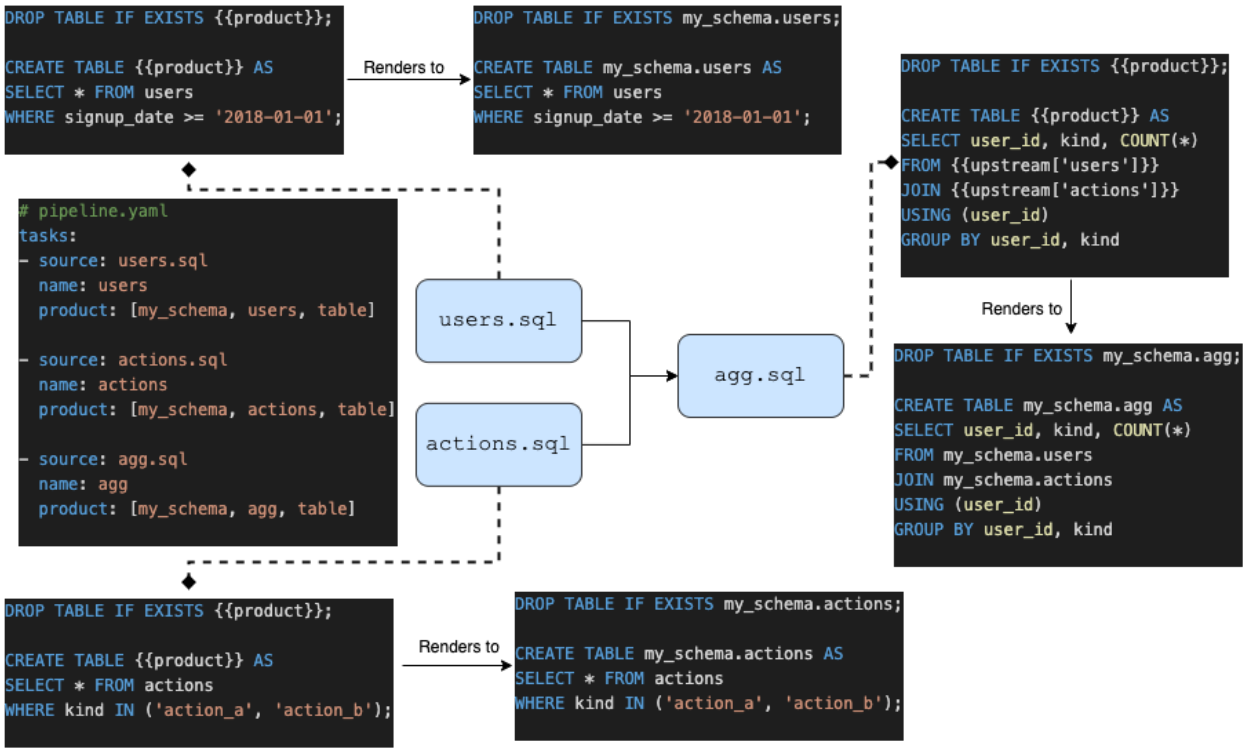
```
ploomber task task_name --source
```

(Change `task_name` for the task you want)

Note: when executing a SQL script, you usually want to replace any existing table/view. Some databases support the `DROP TABLE IF EXISTS` statement to do so, but other databases (e.g., Oracle) have different procedures. Check your database's documentation for details.

Important: Some database drivers do not support sending multiple statements to the database in a single call (e.g., SQLite), in such case, you can use the `split_source` parameter in either `SQLAlchemyClient` or `DBAPIClient` to split your statements and execute them one at a time, allowing you to write a single `.sql` file to perform the `DROP TABLE IF EXISTS` then `CREATE TABLE AS` logic.

The following diagram shows our example pipeline along with some sample source code for each task and the rendered version.



Dumping Data With `SQLDump`

Note: SQLDump only works with *ploomber.clients.SQLAlchemyClient*.

A minimal SQLDump example is available here

If you want to dump the result of a SQL query, use *ploomber.tasks.SQLDump*. Configuring this task is very similar to a regular SQL task:

```
clients
    # client for the database to pull data from
    SQLDump  clients.get_client

tasks
    # some sql tasks here...

    # dump the output of dump-query.sql
    source  sql/dump-query.sql
    # since this is a SQL dump, product is a path to a file
    product  output/data.csv

    # some python tasks here...
```

If you want to dump an entire table, you can do:

```
SELECT * FROM            'some_task'
```

Note that `SQLDump` only works with `SQLAlchemyClient`. Product must be a file with `.csv` or `.parquet` extension.

By default, `SQLDump` downloads data in chunks of 10,000 rows, but yu can change this value:

```
tasks
    source  sql/dump-query.sql
    product  output/data.csv
    # set chunksize to 1 million rows
    chunksize  1000000
```

To dump a single file: `chunksize: null`.

Important: Downloading `.parquet` in chunks may yield errors if the schema inferred from one chunk is not the same as the one in another chunk. If you experience an issue, either change to `.csv` or set `chunksize: null`.

Important: `SQLDump` works with all databases supported by Python because it relies on `pandas` to dump data. However, this introduces a performance overhead. So if you're dumping large tables, consider implementing a solution optimized for your database.

Other SQL Tasks

There are other SQL tasks not covered here, check out the documentation for details:

- *`ploomber.tasks.SQLTransfer`* (move data from one db to another)
- *`ploomber.tasks.SQLDump`* (upload data)
- *`ploomber.tasks.PostgresCopyFrom`* (efficient postgres data upload)

Where To Go From Here

- *SQL templating* shows how to use jinja to write succinct SQL scripts
- Advanced SQL pipeline example
- BigQuery example

To run this locally, install Ploomber and execute: `ploomber examples -n guides/sql-templating`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.3.9 SQL templating

Introductory tutorial teaching how to develop modular SQL pipelines.

Basic templating

SQL templating is a powerful way to make your SQL scripts more concise. It works by using a templating language (Ploomber uses jinja) to generate SQL code on the fly.

You've already used SQL templating if you've followed the SQL pipelines tutorial in the *Get started* section. Let's take a look at the structure of a SQL script in Ploomber:

```
DROP TABLE IF EXISTS

CREATE TABLE             AS
SELECT * FROM            'clean'
WHERE   > 10
```

The `{{product}}` placeholder will be replaced at runtime by whatever value this task has in the `product` section. For example, if you have `product: [schema, name, table]`, `{{product}}` becomes `schema.name`.

To ensure the table is re-created on each run, we add a `DROP TABLE IF EXISTS ...;` statement before `CREATE TABLE ..;`, since both of them take the table name as an argument, we can use the `{{product}}` placeholder.

Finally, the `{{upstream['clean']}}` placeholder tells Ploomber that the current script uses the product from a task named `clean` as input data. This defines the dependency relationship between these two scripts and implies that the placeholder will be replaced by the actual table/view generated by the `clean` task.

These are the essential elements for templated SQL scripts; you don't have to use more if you don't want to but sometimes it is convenient to write concise code and maximize reusability. Let's see a few more examples.

Control structures

jinja offers control structures that help us write SQL code on the fly. Say we want to compute summary statistics on a given column:

```
SELECT

                      as
                        as
                        as
                      as
                      as
                      as
FROM
GROUP BY
```

This code is very repetitive; now imagine how repetitive. We can generate the same code succinctly using a for loop:

```
SELECT

-- loop over aggregation functions
 % for    in  'AVG'  'STDEV'  'COUNT'  'SUM'  'MAX'  'MIN'  %
    -- apply function to the column, name the column
    -- and only add a comma if we are not in the last loop element
                      as                      ',' if not     .last else ''
 %       %
FROM
GROUP BY
```

Macros

Macros let us maximize SQL code reusability by defining snippets that we can "import" into other files. To define a macro, enclose your snippet between the `{% macro MACRO_NAME %} ... {% endmacro %}` tags. Let's create a macro using our previous snippet:

```
# Content of sql/macros.sql
{% macro agg(col_group, col_agg, from_table) -%}

SELECT
    {{col_group}},
{% for fn in ['AVG', 'STDEV', 'COUNT', 'SUM', 'MAX', 'MIN'] %}
    {{fn}}({{col_agg}}) as {{fn}}_{{col_agg}}{{ ',' if not loop.last else '' }}
{% endfor %}
FROM {{from_table}}
GROUP BY {{col_group}}

{%- endmacro %}
```

The `{% macro %}` tag defines the macro name and parameters (if any). To use our macro in a different file, we must import it. Let's say we define the previous macro in a `macros.sql` file:

```
# Content of sql/create-table.sql
-- import macros
{% import "macros.sql" as m %}

DROP TABLE IF EXISTS {{product}};

CREATE TABLE {{product}} AS
-- use macro
{{m.agg(col_group='country', col_agg='price', from_table='sales')}}
```

Configuring support for macros

We have to make a small change to our `pipeline.yaml` file to work with macros. So far, to specify which SQL files to use, we've just passed the file's path in the `source` key. However, to import macros in our scripts, we must configure a source loader.

A source loader is simply a folder with files, with small addition: it defines a "jinja environment" that makes imports work (to know more about jinja environments, click here.

Let's say all the scripts in our pipeline are in a `sql/` directory. `sql/` has two scripts, which correspond to the files shown in the previous section:

[1]:
```
%%sh
tree sql
```

```
sql
├── create-table.sql
└── macros.sql

0 directories, 2 files
```

To configure our source loader. We need to add a `source_loader` section like this:

```
# Content of pipeline.yaml
meta
  # initialize source loader
  source_loader
    # use the sql/ folder as the "root" for loading files
    path  sql/


tasks
  # sources are now loaded from the source loader, paths are relative
  # to the source loader root directory
    source  create-table.sql
    name  sql-task
    product
    client  db.get_client
```

Printing rendered code

Templated SQL helps us write more concise SQL code, but if your template renders to an invalid SQL script, you'll get syntax errors, only use it when the benefits outweigh this risk. One way to debug SQL templates is to see how the rendered code looks like, you can do so from the command line:

[2]:
```
%%sh
ploomber task sql-task --source
```

```
Loading pipeline...
-- import macros


DROP TABLE IF EXISTS some_table;

CREATE TABLE some_table AS
-- use macro
SELECT
    country,

    AVG(price) as AVG_price,

    STDEV(price) as STDEV_price,

    COUNT(price) as COUNT_price,

    SUM(price) as SUM_price,

    MAX(price) as MAX_price,

    MIN(price) as MIN_price

FROM sales
GROUP BY country
```

```
100%|| 1/1 [00:00<00:00, 874.18it/s]
```

As we can see, our template is generating a valid SQL script. But it'd be easier to spot errors in the rendered code than in the templated source if it didn't.

Where to go next

- Jinja documentation

2.3.10 File clients

Note: This is a guide on `file clients`. For API docs see *Clients*.

File clients are used for uploading File products to the cloud. Currently two clients are supported for Amazon S3 and Google Cloud respectively.

During the upload process, an absolute local file path of `/path/to/project/out/data.csv` gets translated to the remote path `path/to/parent/out/data.csv`. Here, `parent` is the parent folder in the bucket to store the files.

Pre-requisites

- Create a bucket in the required cloud platform, or use an existing one.
- Configure the environment with the credentials or create a *credentials.json* file if environment is not configured.

Create a clients file

Next, create a *clients.py* file that contains the below function for S3 client:

```
from                  import

def get_s3
    return                    ='bucket-name'
                          ='parent-folder-name'
                    # pass the json_credentials_path if env not configured with␣
↪credentials
                                        ='credentials.json'
```

Sample file for Google Cloud Storage client:

```
from                  import

def get_gcloud
    return                                ='bucket-name'
                                    ='parent-folder-name'
                                # pass the json_credentials_path if env not configured␣
↪with credentials
                                                    ='credentials.json'
```

Configure the pipeline

Now, configure the *pipeline.yaml* file to add the *clients* key to specify the S3 or GCloud function:

```
# some content
......

# add this
clients
 File  project-name.clients.get_client

 # content continues...
```

Working with external datasets

The file clients only upload products generated by the pipeline. If you want to work with an external dataset, you should download such a dataset in the pipeline task that uses it as input. If you need help contact us on Slack.

Refer: Google cloud template

Note:

- File clients can be used when running pipelines locally as well as when exporting pipelines to external servers (e.g., AWS Batch).
- `ploomber build` commands downloads the existing cloud artifacts for a pipeline run previously.
- The `LocalStorageClient` is mostly used for internal testing and can also be used to locally backup products.

To run this locally, install Ploomber and execute: `ploomber examples -n guides/testing`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.3.11 Pipeline testing

Tutorial showing how to use a task's on_finish hook to test data quality.

Testing your pipeline is critical to ensure your data expectations hold. When you perform a data transformation, you are expecting the output to have certain properties (e.g. no nulls in certain column). Without testing, these expectations won't be verified and will cause errors errors to propagate to all downstream tasks.

These are the most common sources of errors when transforming data:

1. A join operation generates duplicated entries because a wrong assumption of a one-to-one relationship (which is really a one-to-many) in the source tables
2. A function that aggregates data returns `NULL` because at least one of the input data points was `NULL`
3. Dirty data points are used in the analysis (e.g. in a column `age`, you forgot to remove corrupted data points with negative values)

Some of these errors are easy to spot (2), but it might take you some tome to find out about others (1 and 3), or worst, you will never notice these errors and just use incorrect data in your analysis. And even if your code is correct and all your expectations hold true, it might not hold true in the future if the data changes and it's important for you to know this as soon as it happens.

To make testing effective, **your tests should run every time you run your tasks**. Ploomber has a mechanism to automate this.

Sample data

This example loads data from a single table called `my_table`, which has two columns:

1. age: ranges from 21 to 80 but there are some corrupted records with -42
2. score: ranges from 0 to 10 but there are some corrupted records with missing values

Let's take a look at our example `pipeline.yaml`:

[1]:
```
from         import
```

```
# Content of pipeline.yaml
clients
  SQLScript  db.get_client
  SQLDump  db.get_client

tasks
    source  clean.sql
    name  clean
    product   'my_clean_table'  'table'
    on_finish  integration_tests.test_sql_clean

    source  dump.sql
    name  dump
    class  SQLDump
    product  output/my_clean_table.csv
    chunksize  null

    source  transform.py
    product
        nb  output/transformed.html
        data  output/transformed.csv
    on_finish  integration_tests.test_py_transform
```

The pipeline has three tasks, one to clean the raw table, another one to dump the clean data to a CSV file and finally, one Python task to transform the data. We included a SQL and a Python task to show how you can test both types of tasks but we recommend you to do as much analysis as you can using SQL because it scales much better than Python code (you won't have to deal with memory errors).

The configuration is straightforward, the only new key is `on_finish` (inside the first and third task). This is known as a *hook*. Task hooks allow you to embed custom logic when certain events happen. `on_finish` is executed after a task successfully executes. The value is a dotted path, which tells Ploomber where to find your testing function. Under the hood, Ploomber will import your function and call it after the task is executed, here's some equivalent code:

```
from                  import

# your task is executed...

# ploomber calls your testing function...
```

Before diving into the testing source code, let's see the rest of the tasks.

`clean.sql` just filters columns we don't want to include in the analysis:

```
# Content of clean.sql
DROP TABLE IF EXISTS {{product}};

CREATE TABLE {{product}} AS
SELECT * FROM my_table
WHERE score is not null AND age > 0
```

`dump.sql` just selects all rows from the clean table to dump it to the CSV file:

```
# Content of dump.sql
SELECT * FROM {{upstream['clean']}}
```

Finally, the `transform.py` script generates a new column using `score`

```
# Content of transform.py
import        as

# + tags=["parameters"]
         =  'dump'
        = None

# +
   =   .                  'dump'
   'multiplied_score'  =   .      * 42

# +
  .               'data'
```

Let's now take a look at our tests:

```
# Content of integration_tests.py
import        as
from                      import


def test_sql_clean
    """Tests for clean.sql
    """
    assert not                              'score'  'age'
                    =                            'age'
    assert         > 0


def test_py_transform
    """Tests for transform.py
    """
       =   .                       'data'
    assert not   .                  .      .
    assert   .                .      >= 0
```

Testing Python scripts

To test your Python scripts, you have to know which file to look at. You can do so by simply adding `product` as argument to your function. If your Python script generates more than one product (like in our case), `product` will be a dictionary-like object, that's why we are using `product['data']`. This returns a `Product` object, to get the path to the file, simply use the `str` function.

```
>>>         # dictionary-like object: maps names to Product objects
>>>         'data'  # Product object
>>>             'data'   # path to the data file
```

Testing SQL scripts

To test SQL scripts, you also need the client to send queries to the appropriate database, to do so, just add `client` to your testing function.

The `ploomber.testing.sql` module implements convenient functions to test your tables. They always take `client` as its first argument, just pass the client variable directly. Since our SQL script only generates a product, you can directly pass the product object to the testing function (otherwise pass `product[key]`) with the appropriate key.

Note: If you're implementing your own SQL testing logic, doing `str(product)` will return a `{schema}.{name}` string, you can also use `product.schema` and `product.name`.

Running the pipeline

Before we run the pipeline, we generate a sample database:

[2]:
```
%%sh
   setup
python script.py
```

Let's now run our pipeline:

[3]:
```
%%sh
ploomber build
```

```
name       Ran?      Elapsed (s)    Percentage
---------  ------  -------------  ------------
clean      True         0.02013      0.78097
dump       True         0.002051     0.0795713
transform  True         2.55538     99.1395
```

```
Building task 'transform':   0%|          | 0/3 [00:00<?, ?it/s]
Executing:   0%|          | 0/5 [00:00<?, ?cell/s]
Executing:  20%|        | 1/5 [00:01<00:06,  1.75s/cell]
Executing: 100%|| 5/5 [00:02<00:00,  2.39cell/s]
Building task 'transform': 100%|| 3/3 [00:02<00:00,  1.16it/s]
```

Everything looks good.

Let's now imagine a colleague found an error in the cleaning logic and has re-written the script. However, he was unaware that both columns in the raw table had corrupted data and forgot to include the filtering conditions.

The script now looks like this:

[4]:
```
     =      'clean.sql'
         =     .           .        'WHERE score is not null AND age > 0'  ''
    .
```

[4]:
```
86
```

```
# Content of clean.sql
DROP TABLE IF EXISTS {{product}};

CREATE TABLE {{product}} AS
SELECT * FROM my_table
WHERE score is not null AND age > 0
```

Let's see what happens if we run the pipeline:

[5]:
```
%%capture captured
%%   --  -raise-
```

[6]:
```
              .
```

```
Building task 'clean': 100%|| 3/3 [00:00<00:00, 115.03it/s]
Traceback (most recent call last):
  File "/Users/Edu/dev/ploomber/src/ploomber/cli/io.py", line 20, in wrapper
    fn(**kwargs)
  File "/Users/Edu/dev/ploomber/src/ploomber/cli/build.py", line 51, in main
    report = dag.build(force=args.force, debug=args.debug)
  File "/Users/Edu/dev/ploomber/src/ploomber/dag/dag.py", line 482, in build
    report = callable_()
  File "/Users/Edu/dev/ploomber/src/ploomber/dag/dag.py", line 581, in _build
    raise build_exception
  File "/Users/Edu/dev/ploomber/src/ploomber/dag/dag.py", line 513, in _build
    task_reports = self._executor(dag=self,
  File "/Users/Edu/dev/ploomber/src/ploomber/executors/serial.py", line 138, in __call__
    raise DAGBuildError(str(exceptions_all))
ploomber.exceptions.DAGBuildError:
================================ DAG build failed ================================
--------- SQLScript: clean -> SQLRelation(('my_clean_table', 'table')) ---------
---------- /Users/Edu/dev/projects-ploomber/guides/testing/clean.sql ------------
Traceback (most recent call last):
  File "/Users/Edu/dev/ploomber/src/ploomber/tasks/abc.py", line 591, in _build
    self._post_run_actions()
  File "/Users/Edu/dev/ploomber/src/ploomber/tasks/abc.py", line 342, in _post_run_
↪actions
    self._run_on_finish()
  File "/Users/Edu/dev/ploomber/src/ploomber/tasks/abc.py", line 333, in _run_on_finish
    self.on_finish(**kwargs)
  File "/Users/Edu/dev/ploomber/src/ploomber/util/dotted_path.py", line 74, in __call__
    out = self._callable(*args, **kwargs_final)
  File "/Users/Edu/dev/projects-ploomber/guides/testing/integration_tests.py", line 8,␣
↪in test_sql_clean
    assert not nulls_in_columns(client, ['score', 'age'], product)
AssertionError
```

(continues on next page)

(continued from previous page)

```
The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/Users/Edu/dev/ploomber/src/ploomber/executors/serial.py", line 186, in catch_
↪exceptions
    fn()
  File "/Users/Edu/dev/ploomber/src/ploomber/executors/serial.py", line 159, in __call__
    return self.fn(**self.kwargs)
  File "/Users/Edu/dev/ploomber/src/ploomber/executors/serial.py", line 166, in catch_
↪warnings
    result = fn()
  File "/Users/Edu/dev/ploomber/src/ploomber/executors/serial.py", line 159, in __call__
    return self.fn(**self.kwargs)
  File "/Users/Edu/dev/ploomber/src/ploomber/executors/serial.py", line 235, in build_in_
↪subprocess
    report, meta = task._build(**build_kwargs)
  File "/Users/Edu/dev/ploomber/src/ploomber/tasks/abc.py", line 603, in _build
    raise TaskBuildError(msg) from e
ploomber.exceptions.TaskBuildError: Exception when running on_finish for task "clean":
============================== Summary (1 task) ==============================
SQLScript: clean -> SQLRelation(('my_clean_table', 'table'))
============================== DAG build failed ==============================
```

Ploomber a structured error message to understand why your pipeline failed. The last few lines are a summary:

```
============================== Summary (1 task) ==============================
SQLScript: clean -> SQLRelation(('my_clean_table', 'table'))
============================== DAG build failed ==============================
```

By looking at the summary we know our pipeline failed because one task crashed (`clean`). If we scroll up we'll see a header section:

```
--------- SQLScript: clean -> SQLRelation(('my_clean_table', 'table')) ---------
-------------- /Users/Edu/dev/projects-ploomber/testing/clean.sql --------------
```

Each task displays its traceback on a separate section. Since only one task failed in our example we only see one task traceback.

At the end of this task traceback, we see the following line:

```
Exception when running on_finish for task "clean":
```

Now we know that the `on_finish` hook crashed. If we go up a few lines up:

```
assert not nulls_in_columns(client, ['score', 'age'], product)
AssertionError
```

That tells me the exact test that failed! Pipelines can get very large; it helps a lot to have a structured error message that tells us what failed and where it happened. Our take away from the error message is: "the pipeline building process

failed because the `on_finish` hook in the `clean` task raised an exception in certain assertion". That's much better than either "the pipeline failed" or "this line raised an exception".

Let's fix our pipeline and add the `WHERE` clause again:

[7]:
```
     =      'clean.sql'
         =     .            + 'WHERE score is not null AND age > 0'


    .
```

```
DROP TABLE IF EXISTS {{product}};

CREATE TABLE {{product}} AS
SELECT * FROM my_table
WHERE score is not null AND age > 0
```

[7]:
```
121
```

[8]:
```
%%sh
ploomber build
```

```
name       Ran?      Elapsed (s)    Percentage
---------  ------  -------------  ------------
clean      True         0.017694     0.67259
dump       True         0.001669     0.0634426
transform  True         2.61136     99.264
```

```
Building task 'transform':   0%|          | 0/3 [00:00<?, ?it/s]
Executing:   0%|          | 0/5 [00:00<?, ?cell/s]
Executing:  20%|        | 1/5 [00:01<00:07,  1.78s/cell]
Executing: 100%|| 5/5 [00:02<00:00,  2.34cell/s]
Building task 'transform': 100%|| 3/3 [00:02<00:00,  1.13it/s]
```

All good! Pipeline is running without issues again!

Test-driven development (TDD)

Writing data tests is essential for developing robust pipelines. Coding tests is simple, all we have to do is write in code that we already have in our mind when thinking what the outcome of a script should be.

This thought process happens *before* we write the actual code, which means we could easily write tests even before we write the actual code. This approach is called Test-driven development (TDD).

Following this framework has an added benefit, since we force ourselves to put in concrete terms our data expectations, it makes easier to think how we want to get there.

Furthermore, *tests also serve as documentation* for us (and for others). By looking at our tests, anyone can see what *our intent* is. Then by looking at the code, it will be easier to spot mismatches between our intent and our implementation.

Pro tip: debugging and developing tests interactively

Even though tests are usually just a few short statements, writing them in an interactive way can help you quickly prototype your assertions. One simple trick you can use to do this is to start an interactive session and load the `client` and `product` variables.

Let's imagine you want to write a test for a new SQL script (but the same applies for other types of scripts). You add a testing function, but it's currently empty:

```
def my_sql_testing_function
    pass
```

If you run this, Ploomber will still call your function, you can start an interactive session when this happens:

```
def my_sql_testing_function
    from         import
```

Once you call `ploomber build`, wait for the Python prompt to show and verify you have the `client` and `product` variables:

```
>>>
>>>
```

Where to go next

- *Documentation for ploomber.testing - Handy functions for testing pipelines*
- Our blog post on CI for Data Science (which includes a section on testing pipelines)

To run this locally, install Ploomber and execute: `ploomber examples -n guides/debugging`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.3.12 Debugging

Tutorial showing techniques for debugging pipelines.

For a quick reference, click here.

Debugger basics

Skip this if you're already familiar with the Python debugger.

A debugger is a program that helps inspect another program for debugging. Python comes with a debugger called pdb.

There are a few approaches for debugging programs. One approach is line-by-line debugging, which starts our program in *debug* mode so we can easily inspect variables, move to the next line, etc.

One important concept to know when debugging is *stack frame*. Simply speaking, stack frames represent the state of our code at a given level. When you write a non-trivial function, it will depend on other functions to work (yours or from third party packages). Each function has its own stack frame which defines the variables that are available to it.

When a program fails, it can do so at different levels (i.e. a different stack frame). Let's see a simple example:

```
def reciprocal
    return 1/

def reciprocal_and_multiply
    return               *
```

There are two places where things can go wrong in the program above: if we pass `x=0`, the `reciprocal` operation will fail. If we pass `y=None`, the program fails, but it will do so in the `reciprocal_and_multiply` function. For this trivial example, it's easy to see at which level the code breaks but in a real program the source code alone is usually not enough to know. Moving between stack frames can help you find out where the error is coming from.

Understanding error messages

Let's take a look at our example pipeline declaration:

```
# Content of pipeline.yaml
tasks
    source  load.py
    product
      nb  output/raw.html
      train  output/train.csv
      test  output/test.csv

    source  preprocess.py
    product  output/clean.html
```

Very simple, two tasks. One loads the data and the next one preprocess it.

Let's run the pipeline and then analyze the output:

[1]:
```
%%sh --no-raise-error
ploomber build --force
```

```
Loading pipeline...
Building task 'load':   0%|          | 0/2 [00:00<?, ?it/s]
Executing:   0%|          | 0/5 [00:00<?, ?cell/s]
Executing:  20%|        | 1/5 [00:01<00:07,  1.97s/cell]
Executing: 100%|| 5/5 [00:02<00:00,  2.15cell/s]
Building task 'preprocess':  50%|     | 1/2 [00:02<00:02,  2.86s/it]
Executing:   0%|          | 0/6 [00:00<?, ?cell/s]
Executing:  17%|        | 1/6 [00:03<00:15,  3.15s/cell]
Executing:  67%|   | 4/6 [00:03<00:01,  1.60cell/s]
Executing: 100%|| 6/6 [00:03<00:00,  1.57cell/s]
Building task 'preprocess': 100%|| 2/2 [00:06<00:00,  3.41s/it]

================================ DAG build failed ================================
----------- NotebookRunner: preprocess -> File('output/clean.html') ------------
------- /Users/Edu/dev/projects-ploomber/guides/debugging/preprocess.py --------

--------------------------------------------------------------------------------
Exception encountered at "In [6]":
--------------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
/var/folders/3h/_lvh_w_x5g30rrjzb_xnn2j80000gq/T/ipykernel_66697/1567702903.py in
↪<module>
----> 1 my_preprocessing_function(X_train, X_test)

/var/folders/3h/_lvh_w_x5g30rrjzb_xnn2j80000gq/T/ipykernel_66697/2058553394.py in my_
↪preprocessing_function(X_train, X_test)
      2     encoder = OneHotEncoder()
      3     X_train_t = encoder.fit_transform(X_train)
----> 4     X_test_t = encoder.transform(X_test)
      5     return X_train_t, X_test_t

~/miniconda3/envs/projects/lib/python3.9/site-packages/sklearn/preprocessing/_encoders.
↪py in transform(self, X)
    880             "infrequent_if_exist",
    881         }
--> 882         X_int, X_mask = self._transform(
    883             X,
    884             handle_unknown=self.handle_unknown,

~/miniconda3/envs/projects/lib/python3.9/site-packages/sklearn/preprocessing/_encoders.
↪py in _transform(self, X, handle_unknown, force_all_finite, warn_on_unknown)
    158                         " during transform".format(diff, i)
    159                     )
--> 160                     raise ValueError(msg)
    161                 else:
    162                     if warn_on_unknown:

ValueError: Found unknown categories ['d'] in column 0 during transform

ploomber.exceptions.TaskBuildError: ("Error when executing task 'preprocess'. Partially␣
↪executed notebook available at /Users/Edu/dev/projects-ploomber/guides/debugging/
↪output/clean.ipynb", NotebookRunner: preprocess -> File('output/clean.html'))
ploomber.exceptions.TaskBuildError: Error building task "preprocess"
============================== Summary (1 task) ==============================
NotebookRunner: preprocess -> File('output/clean.html')
============================== DAG build failed ==============================

Need help? https://ploomber.io/community
```

The summary at the bottom gives a high-level explanation:

```
============================== Summary (1 task) ==============================
NotebookRunner: preprocess -> File('output/clean.html')
============================== DAG build failed ==============================
```

The `preprocess` task failed. Go up a few lines:

```
ploomber.exceptions.TaskBuildError: An error occurred when calling papermil.execute_
↪notebook, partially executed notebook with traceback available at ...
```

That's useful, it tells us where we can find the partially executed notebook in case we want to take a look at it. A few lines up:

```
ValueError: Found unknown categories ['d'] in column 1 during transform
```

That's the exact line that failed, if you take a look at the original error traceback, you'll see that the actual line that raised the exception comes from the scikit-learn library (`_encoders.py` file):

```
~/miniconda3/envs/ploomber/lib/python3.6/site-packages/sklearn/preprocessing/_encoders.
↪py in _transform(self, X, handle_unknown)
    122                     msg = ("Found unknown categories {0} in column {1}"
    123                            " during transform".format(diff, i))
--> 124                     raise ValueError(msg)
    125                 else:
    126                     # Set the problematic rows to an acceptable value and

ValueError: Found unknown categories ['d'] in column 0 during transform
```

The error message provides us a lot of information: Our pipeline failed while executing task `preprocess`. Somewhere in our task's code we ran something that made scikit-learn crash.

Let's take a look at the failing task's source code:

```
# Content of preprocess.py
# %%
import        as
from                      import

# %% tags=["parameters"]
         =  'load'
        = None



# %%
def my_preprocessing_function
            =
              =        .
             =        .
    return


# %%
        =   .                  'load'  'train'
       =   .                  'load'  'test'

# %%
```

Our `preprocess.py` script is using scikit-learn's `OneHotEncoder` to transform variables. The error message offers some information but not enough to fix the issue (*we don't have a column named "0"!*). There must be something going on internally.

This is a good use case for Ploomber's debugging capabilities.

Starting a debugging session

There are three ways to start a debugger:

1. Jump to the first line and start the debugger
2. (post-mortem) Let the task run and start the debugger as soon as it fails
3. (breakpoint) Jump to a specific line and start the debugger

We'll analyze the three of them but feel free to jump to the one that's more applicable to your use case.

Jump to the first line and start the debugger

To start a debugging session you first have to start an interactive session. To do so, run the following command in the terminal:

```
ploomber interact
```

When it finishes setting things up, your pipeline will be available in the `dag` variable. This is a standard Python session, you can execute any Python code you want.

We already know that the error is happening in the `preprocess` task, you can start a line-by-line debugging session with the following command:

```
>>>     'preprocess' .
```

Here's a replay of my debugging session (with comments):

```
# COMMENT: I entered the command "next" a few times until I reached the failing line
ipdb>
> /var/folders/3h/_lvh_w_x5g30rrjzb_xnn2j80000gq/T/tmpbatitar6.py(45)<module>()
     41 X_train = pd.read_csv(upstream['load']['train'])
     42 X_test = pd.read_csv(upstream['load']['test'])
     43
     44 # + tags=[]
# COMMENT: "--->" means that line will be executed when I send the "next" command
---> 45 my_preprocessing_function(X_train, X_test)

ipdb>
# COMMENT: Same error message that we got before!
ValueError: Found unknown categories ['d'] in column 0 during transform
> /var/folders/3h/_lvh_w_x5g30rrjzb_xnn2j80000gq/T/tmpbatitar6.py(45)<module>()
     41 X_train = pd.read_csv(upstream['load']['train'])
     42 X_test = pd.read_csv(upstream['load']['test'])
     43
     44 # + tags=[]
---> 45 my_preprocessing_function(X_train, X_test)
# COMMENT: I entered the "down" command to move one stack frame down (inside my_
↪preprocessing_function)
ipdb> down
> /var/folders/3h/_lvh_w_x5g30rrjzb_xnn2j80000gq/T/tmpbatitar6.py(36)my_preprocessing_
↪function()
     34     encoder = OneHotEncoder()
     35     X_train_t = encoder.fit_transform(X_train)
```

(continues on next page)

(continued from previous page)

```
# COMMENT: The line that raised the exception
---> 36     X_test_t = encoder.transform(X_test)
     37     return X_train_t, X_test_t
     38

# COMMENT: Print X_train and X_test
ipdb> X_train
  cat
0   a
1   b
2   c
ipdb> X_test
  cat
0   a
1   b
2   d
# COMMENT: Exit debugger with the "quit" command
ipdb> quit
```

Ah-ha! The encoder is fitted with a column with values `a`, `b` and `c` but then applied to a testing set with value `d`. That's why it's breaking.

This is an example of how your code could be doing everything right, but your data is incompatible. How you fix this is up to us. The important thing is that we know why things are failing.

(post-mortem) Let the task run and start the debugger as soon as it fails

Note: post-mortem debugging was improved in Ploomber 0.20, ensure have at least that version

An alternative approach is to let the program run and start the debugging session as soon as it finds an exception, this is called *post-mortem* debugging.

To start a debugging session"

```
ploomber task {task-name} --debug
```

or:

```
ploomber build --debug
```

This will start the debugger as soon as the code breaks, alternatively, you can serialize the error to start the debugger session whenever you want:

Added in Ploomber 0.20

```
ploomber task {task-name} --debuglater
```

or:

```
ploomber build --debuglater
```

Then, start the debugging session with:

```
dltr {task-name}.dump
```

Note: you may delete the ``{task-name}.dump``file once you are done debugging

Important: Beware that using ``–debuglater`` will serialize all the variables, so ensure you have enough disk space when using it, especially if running with the Parallel executor

Here's the (commented) replay of my post-mortem debugging session:

```
# COMMENT: I deleted a few lines for brevity
ValueError: Found unknown categories ['d'] in column 0 during transform
> /Users/Edu/miniconda3/envs/ploomber/lib/python3.6/site-packages/sklearn/preprocessing/_
↪encoders.py(124)_transform()
    122                     msg = ("Found unknown categories {0} in column {1}"
    123                            " during transform".format(diff, i))
# COMMENT: The session starts here. Not very useful because we are inside the scikit-
↪learn package
# (note the file path: site-packages/sklearn/preprocessing/_encoders.py)
--> 124                     raise ValueError(msg)
    125                 else:
    126                     # Set the problematic rows to an acceptable value and
# COMMENT: Let's move up
ipdb> up
> /Users/Edu/miniconda3/envs/ploomber/lib/python3.6/site-packages/sklearn/preprocessing/_
↪encoders.py(428)transform()
    426         check_is_fitted(self)
    427         # validation of X happens in _check_X called by _transform
--> 428         X_int, X_mask = self._transform(X, handle_unknown=self.handle_unknown)
    429
    430         n_samples, n_features = X_int.shape
# COMMENT: Still inside scikit-learn, let's move up again
ipdb> up
> /var/folders/3h/_lvh_w_x5g30rrjzb_xnn2j80000gq/T/tmp653y199s.py(36)my_preprocessing_
↪function()
     34     encoder = OneHotEncoder()
     35     X_train_t = encoder.fit_transform(X_train)
# COMMENT: Now are are in our task's code, same place as in the previous example
---> 36     X_test_t = encoder.transform(X_test)
     37     return X_train_t, X_test_t
     38
ipdb> X_train
  cat
0   a
1   b
2   c
ipdb> X_test
  cat
0   a
1   b
2   d
ipdb> quit
```

As you can see, we can use either of these two approaches.

(breakpoint) Jump to a specific line and start the debugger

The previous example showed how we could debug a program that raises an exception. A more difficult scenario is when our program runs without errors but we find issues in the output (e.g. charts are not displaying correctly, data file has NAs, etc).

This is a much harder problem because we don't know where to look at! If a bug is originated in task `A` it might propagate to any downstream tasks that use the product from `A` as input, this is why testing is essential. By explicitly checking our data expectations, we increase the chance of catching errors at the source, rather than in a downstream task.

When it happens (and trust me, it will), we recommend you to follow a recursive approach: Once you detect the error, the first question to answer is: Which task produced this output? Once you know that, start a line-by-line debugging session (post-mortem won't work because there is no exception!), and carefully check variables to see if you can spot the error.

If everything looks correct, go to all upstream tasks and repeat this process. You can do this from the command line.

First, start an interactive session from the terminal:

```
ploomber interactive
```

Then debug the task that produced the buggy output:

```
>>>     'buggy_task' .
```

If that's not enough, check upstream tasks. To find upstream tasks, use `task.upstream`:

```
>>>     'buggy_task' .
```

If you have a hypothesis of *where the error might be*. You can insert a breakpoint in your task's source code to start a debugging session at any given point:

```
# buggy_task.py

# some code
# ...
# ...

# breakpoint: this is where I think the bug is...
import          .


# more code
# ...
# ...
```

Then start a post-mortem session. The debugger will start at the line where you inserted the breakpoint.

Using the CLI to check if we fixed the bug

In a real scenario, we might try a few things before we find bug fix. To quickly iterate over candidate solutions, we'd like to check if the applied change makes our pipeline *not* to throw an error. This is where Ploomber's incremental builds come in handy.

If we narrowed down the error to a specific task, we can apply changes and quickly check if the new code runs correctly by just running that task:

```
ploomber task {task-name}
```

If the exception happens in task `B`, but the solution has to be implemented in task `A` (where `A` is an upstream dependency of `B`), then we have to make sure that we run `A` and `B` to verify the fix. A full end-to-end run is wasteful but so is an incremental run if `B` has many downstream tasks. For testing purposes, we just care about things going well *until* ``B``. This is a good use of a partial build: it will run all tasks until it reaches a selected task (by default, it will still skip up-to-date tasks). In our case:

```
ploomber build --partially B
```

Letting our pipeline fail under unforeseen circumstances

The error in our program is of particular interest because it posits a common scenario: our program is correct but still failed due to unforeseen circumstances (unexpected data properties). Although these bugs challenge our assumptions about input data, fixing the error is just as important as explaining *why* we fixed the way we did it.

Picture this: we decide to drop all observations that contain the unexpected value (`d`), now our pipeline runs correctly. A few months later, we receive new data so we run the pipeline again, but we run into the same issue because of a new unexpected value (say, `e`).

We could argue that one solution would be to *drop all unexpected values*. Is this the best approach? Dropping observations silently is dangerous, as they might contain helpful information for our analysis. If we bury a `drop=True` piece of code in a pipeline with dozens of files, we will cause *a lot of* trouble to someone (which could be us) in the future. As we mentioned in the previous guide: explicitly stating our data expectations is the way to move forward.

If we decide dropping `d` is a reasonable choice, we can encode our new data expectations in the upstream task testing function (because that's the task that supplies input data). Let's recall how our pipeline looks like:

```
# Content of pipeline.yaml
tasks
    source  load.py
    product
      nb  output/raw.html
      train  output/train.csv
      test  output/test.csv

    source  preprocess.py
    product  output/clean.html
```

`load` supplies input for `preprocess`. Our testing function for the `load` task would look like this:

```
def test_load
          =   .                    'train'
         =   .                   'test'

    # NOTE: these are the expected values in the cat column
```

(continues on next page)

(continued from previous page)

```
    # we expect value 'd' in the testing set and we'll
    # drop it during preprocessing. Any other unexpected
    # values will raise an exception here so we have the
    # chance to decide what to do with it
    assert           'cat' .          == 'a'  'b'  'c'
    assert          'cat' .          == 'a'  'b'  'c'  'd'
```

The comment should actually be part of the testing function, without it, there is no context to understand why are we testing such a specific condition.

Debugging (templated) SQL scripts

So far, we've discussed how to debug Python scripts, but SQL scripts can also fail. In a previous guide, we showed how templated SQL scripts help us write more concise SQL code, but this comes with a cost. Relying too much on templating makes our templated source code short but hard to read. If your database complains about syntax errors when executing SQL tasks, chances are, the errors is coming from incorrect templating logic. One good first debugging step is to take a look at the rendered code. You can do so from the command line:

```
ploomber task {task-name} --source
```

Apart from looking at rendered code, there isn't much to say about debugging SQL scripts because there are no interactive debuggers. The best we can do is to organize our scripts in a clear way to make it easy to spot errors.

Where to go next

- pdb documentation

To run this locally, install Ploomber and execute: `ploomber examples -n guides/versioning`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.3.13 Versioning

Note: This feature requires Ploomber 0.17.1 or higher.

A tutorial showing how to version pipeline products.

Although Ploomber is not a data versioning solution, it offers a simple way to organize pipeline artifacts via placeholders. Note that this requires your project to be in a git repository.

Using `{{git}}`

Let's look at the first example, which uses the `{{git}}` placeholder:

```
# Content of pipeline.git.yaml
tasks
    source  tasks/load.py
    product
      nb  'output/{{git}}/load.html'
      data  'output/{{git}}/data.csv'
```

(continues on next page)

(continued from previous page)

```
    source  tasks/plot.py
    product
      nb  'output/{{git}}/plot.html'
```

You can see that both tasks use `{{git}}`. When Ploomber executes the pipeline, it will replace the placeholder using the following order:

1. If currently at the tip of the branch, return the branch name
2. If the current commit has a tag, return the tag name
3. Otherwise, return the hash for the current commit (appending `-dirty` if there are uncommitted changes)

Let's see how it works:

[1]:
```
from         import

from               import
```

[2]:
```
    =         "pipeline.git.yaml" .
    "load" .        "nb"
```

[2]:
```
File('output/master/load.html')
```

We can see the product will be stored in the `output/master` directory, `{{git}}` is resolved to `master` since we're at the tip of such branch.

Using `{{git_hash}}`

The `{{git_hash}}` placeholder is similar to `{{git}}`, except it doesn't return the branch name, the rules are as follows:

1. If the current commit has a tag, return the tag name
2. Otherwise, return the hash for the current commit (appending `-dirty` if there are uncommitted changes)

This is how our sample `pipeline.git_hash.yaml` looks like:

```
# Content of pipeline.git_hash.yaml
tasks
    source  tasks/load.py
    product
      nb  'output/{{git_hash}}/load.html'
      data  'output/{{git_hash}}/data.csv'

    source  tasks/plot.py
    product
      nb  'output/{{git_hash}}/plot.html'
```

[3]:
```
    =         "pipeline.git_hash.yaml" .
    "load" .        "nb"
```

[3]:
```
File('output/62a3494-dirty/load.html')
```

This time, the product will be stored in a directory with the hash of the current commit.

Adding the current timestamp with {{now}}

Alternatively, you can use the {{now}} placeholder, which doesn't require your project to be in a git repository and will resolve to the current timestamp:

```
# Content of pipeline.now.yaml
tasks
    source  tasks/load.py
    product
      nb  'output/{{now}}/load.html'
      data  'output/{{now}}/data.csv'

    source  tasks/plot.py
    product
      nb  'output/{{now}}/plot.html'
```

[4]:
```
    =         "pipeline.now.yaml" .

     =          "load" .        "nb"  .                 .
```

```
output/2022-03-26T17:00:38.060493/load.html
```

You can see that the load.html file will to into a folder with the timestamp computed when running this example.

Using placeholders in selected tasks

You can selectively choose which tasks to organize based on the git repository commit, the following example only uses the {{git}} placeholder in the last task:

```
# Content of pipeline.partial.yaml
tasks
    source  tasks/load.py
    product
      nb  'output/load.html'
      data  'output/data.csv'

    source  tasks/plot.py
    product
      nb  'output/{{git}}/plot.html'
```

[5]:
```
    =         "pipeline.partial.yaml" .
    "load" .        "nb"
```

[5]:
```
File('output/load.html')
```

[6]:
```
    "plot" .        "nb"
```

[6]:
```
File('output/master/plot.html')
```

Here, you can see that the product of the load task goes to output/, while the output of plot goes to output/master/

Using an `env.yaml`

If you're using an `env.yaml` file, you can still use the placeholders:

```
# env.yaml
directory  '{{git}}' # or '{{git_hash}}'
```

Then add references to `{{directory}}` in your `pipeline.yaml`:

```
# pipeline.yaml
tasks
    source  tasks/load.py
    product
      nb  'output/{{directory}}/load.html'
      data  'output/{{directory}}/data.csv'
```

To run this locally, install Ploomber and execute: `ploomber examples -n guides/logging`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.3.14 Logging

Tutorial showing how to add logging to a pipeline.

Sample pipeline

The pipeline we'll be using for this guide contains two tasks (a script and a function):

```
# Content of basic/pipeline.yaml
tasks
    source  script.py
    product  output/nb-log.html
    papermill_params
      log_output  True

    source  tasks.function
    product  output/fn-log.txt
```

Note that the script task contains:

```
papermill_params
    log_output  True
```

This extra configuration is required on each script/notebook task in your pipeline to enable logging. The code on each task isn't important; they contain a for loop and log a message on each iteration. Let's see it in action:

[1]:
```
%%sh
   basic
ploomber build --log info --force
```

```
Loading pipeline...
name      Ran?      Elapsed (s)    Percentage
--------  ------  -------------  ------------
script    True          4.66689       60.8412
function  True          3.00372       39.1588
```

```
WARNING:traitlets:Kernel Provisioning: The 'local-provisioner' is not found.  This is␣
↪likely due to the presence of multiple jupyter_client distributions and a previous␣
↪distribution is being used as the source for entrypoints - which does not include
↪'local-provisioner'.  That distribution should be removed such that only the version-
↪appropriate distribution remains (version >= 7).  Until then, a 'local-provisioner'␣
↪entrypoint will be automatically constructed and used.
The candidate distribution locations are: ['/workspaces/projects/venv/lib/python3.8/site-
↪packages/jupyter_client-7.3.2.dist-info']
INFO:blib2to3.pgen2.driver:Generating grammar tables from /workspaces/projects/venv/lib/
↪python3.8/site-packages/blib2to3/Grammar.txt
INFO:blib2to3.pgen2.driver:Writing grammar tables to /home/codespace/.cache/black/22.3.0/
↪Grammar3.8.13.final.0.pickle
INFO:blib2to3.pgen2.driver:Writing failed: [Errno 2] No such file or directory: '/home/
↪codespace/.cache/black/22.3.0/tmp1j9p6bb9'
INFO:blib2to3.pgen2.driver:Generating grammar tables from /workspaces/projects/venv/lib/
↪python3.8/site-packages/blib2to3/PatternGrammar.txt
INFO:blib2to3.pgen2.driver:Writing grammar tables to /home/codespace/.cache/black/22.3.0/
↪PatternGrammar3.8.13.final.0.pickle
INFO:blib2to3.pgen2.driver:Writing failed: [Errno 2] No such file or directory: '/home/
↪codespace/.cache/black/22.3.0/tmp2rs4570f'
INFO:ploomber.dag.dag:Building DAG DAG("basic")
Building task 'script':   0%|          | 0/2 [00:00<?, ?it/s]INFO:ploomber.tasks.abc.
↪NotebookRunner:Starting execution: NotebookRunner: script -> File('output/nb-log.html')
INFO:papermill:Input Notebook:  /tmp/tmp_odbqsk5.ipynb
INFO:papermill:Output Notebook: /workspaces/projects/guides/logging/basic/output/nb-log.
↪ipynb

Executing:   0%|          | 0/6 [00:00<?, ?cell/s]INFO:papermill:Executing notebook with␣
↪kernel: python3
INFO:papermill:Executing Cell 1---------------------------------------
INFO:papermill:Ending Cell 1-----------------------------------------

Executing:  17%|        | 1/6 [00:01<00:05,  1.02s/cell]INFO:papermill:Executing Cell 2--
↪-------------------------------------
INFO:papermill:Ending Cell 2-----------------------------------------
INFO:papermill:Executing Cell 3---------------------------------------
INFO:papermill:Ending Cell 3-----------------------------------------
INFO:papermill:Executing Cell 4---------------------------------------
INFO:papermill:Ending Cell 4-----------------------------------------
INFO:papermill:Executing Cell 5---------------------------------------
INFO:papermill:Ending Cell 5-----------------------------------------
INFO:papermill:Executing Cell 6---------------------------------------
INFO:papermill:INFO:__main__:[script log] Finished step 1...

INFO:papermill:INFO:__main__:[script log] Finished step 2...

INFO:papermill:INFO:__main__:[script log] Finished step 3...
```

(continues on next page)

(continued from previous page)

```
INFO:papermill:INFO:__main__:[script log] Done.

INFO:papermill:Ending Cell 6----------------------------------------

Executing: 100%|| 6/6 [00:04<00:00,  1.40cell/s]
INFO:ploomber.tasks.abc.NotebookRunner:Done. Operation took 4.7 seconds
Building task 'function':  50%|     | 1/2 [00:04<00:04,  4.67s/it]INFO:ploomber.tasks.
↪abc.PythonCallable:Starting execution: PythonCallable: function -> File('output/fn-log.
↪txt')
INFO:tasks:[function log] Finished step 1...
INFO:tasks:[function log] Finished step 2...
INFO:tasks:[function log] Finished step 3...
INFO:tasks:[function log] Done.
INFO:ploomber.tasks.abc.PythonCallable:Done. Operation took 3.0 seconds
Building task 'function': 100%|| 2/2 [00:07<00:00,  3.85s/it]
INFO:ploomber.dag.dag: DAG report:
name      Ran?      Elapsed (s)    Percentage
--------  ------  -------------  ------------
script    True          4.66689       60.8412
function  True          3.00372       39.1588
```

We can see that the logging statements appear in the console. If you want to take a look at the code, click here.

Why not print?

Note that the snippets above use the `logging` module instead of `print`. Although `print` is a quick and easy way to display messages in the console, the `logging` module is more flexible. Hence, it is the recommended option.

Logging to a file

It's common to send all your log records to a file. You can do so with the `--log-file/-F` option:

```
ploomber build --log info --log-file my.log
```

Logging to a file from Python

Alternatively, you can configure logging from Python, which gives you more flexibility:

[2]:
```
# you may store the contents of this cell in a .py file and then call it from the␣
↪command line
# e.g., python run_pipeline.py
import
from         import
import

from                    import
from               import

if      'my.log' .
```

(continues on next page)

(continued from previous page)

```
        'my.log' .

      .                   ='my.log'       ='%(levelname)s:%(message)s'       =       .
↪

# make sure we can import basic/tasks.py since basic/pipeline.yaml uses it
   .    .       'basic'

    =         'basic/pipeline.yaml' .
   .           =True
```

```
  0%|          | 0/2 [00:00<?, ?it/s]
```

```
Executing:   0%|          | 0/6 [00:00<?, ?cell/s]
```

[2]:
```
name      Ran?      Elapsed (s)    Percentage
--------  ------  -------------  ------------
script    True          4.76759       61.3428
function  True          3.00445       38.6572
```

Let's look at the file contents:

[3]:
```
          'my.log' .
```

```
INFO:Generating grammar tables from /workspaces/projects/venv/lib/python3.8/site-
↪packages/blib2to3/Grammar.txt
INFO:Writing grammar tables to /home/codespace/.cache/black/22.3.0/Grammar3.8.13.final.0.
↪pickle
INFO:Writing failed: [Errno 2] No such file or directory: '/home/codespace/.cache/black/
↪22.3.0/tmpeca0cxtc'
INFO:Generating grammar tables from /workspaces/projects/venv/lib/python3.8/site-
↪packages/blib2to3/PatternGrammar.txt
INFO:Writing grammar tables to /home/codespace/.cache/black/22.3.0/PatternGrammar3.8.13.
↪final.0.pickle
INFO:Writing failed: [Errno 2] No such file or directory: '/home/codespace/.cache/black/
↪22.3.0/tmp3djgccxy'
INFO:Building DAG DAG("basic")
INFO:Starting execution: NotebookRunner: script -> File('basic/output/nb-log.html')
INFO:Input Notebook:  /tmp/tmp1y05qp_7.ipynb
INFO:Output Notebook: /workspaces/projects/guides/logging/basic/output/nb-log.ipynb
INFO:Executing notebook with kernel: python3
INFO:Executing Cell 1---------------------------------------
INFO:Ending Cell 1-------------------------------------------
INFO:Executing Cell 2---------------------------------------
INFO:Ending Cell 2-------------------------------------------
INFO:Executing Cell 3---------------------------------------
INFO:Ending Cell 3-------------------------------------------
INFO:Executing Cell 4---------------------------------------
INFO:Ending Cell 4-------------------------------------------
INFO:Executing Cell 5---------------------------------------
INFO:Ending Cell 5-------------------------------------------
INFO:Executing Cell 6---------------------------------------
INFO:INFO:__main__:[script log] Finished step 1...
```

(continues on next page)

(continued from previous page)

```
INFO:INFO:__main__:[script log] Finished step 2...

INFO:INFO:__main__:[script log] Finished step 3...

INFO:INFO:__main__:[script log] Done.

INFO:Ending Cell 6----------------------------------------
INFO:Done. Operation took 4.8 seconds
INFO:Starting execution: PythonCallable: function -> File('basic/output/fn-log.txt')
INFO:[function log] Finished step 1...
INFO:[function log] Finished step 2...
INFO:[function log] Finished step 3...
INFO:[function log] Done.
INFO:Done. Operation took 3.0 seconds
INFO: DAG report:
name      Ran?      Elapsed (s)    Percentage
--------  ------  -------------  ------------
script    True          4.76759       61.3428
function  True          3.00445       38.6572
```

Controlling logging level

The Python's logging module allows you to filter messages depending on their priority. For example, when running your pipeline, you may only want to display *regular* messages, but you may allow *regular* and *debugging* messages for more granularity when debugging. Since Ploomber runs tasks differently depending on their type (i.e., functions vs. scripts/notebooks), controlling the logging level requires a bit of extra work. Let's use the same pipeline in the `parametrized` directory:

[4]:
```
%%sh
   parametrized
ploomber build --log info --env--logging_level info --force
```

```
Loading pipeline...
name      Ran?      Elapsed (s)    Percentage
--------  ------  -------------  ------------
script    True          4.49249       59.9239
function  True          3.0045        40.0761
```

```
WARNING:traitlets:Kernel Provisioning: The 'local-provisioner' is not found.  This is␣
↪likely due to the presence of multiple jupyter_client distributions and a previous␣
↪distribution is being used as the source for entrypoints - which does not include
↪'local-provisioner'.  That distribution should be removed such that only the version-
↪appropriate distribution remains (version >= 7).  Until then, a 'local-provisioner'␣
↪entrypoint will be automatically constructed and used.
The candidate distribution locations are: ['/workspaces/projects/venv/lib/python3.8/site-
↪packages/jupyter_client-7.3.2.dist-info']
INFO:blib2to3.pgen2.driver:Generating grammar tables from /workspaces/projects/venv/lib/
↪python3.8/site-packages/blib2to3/Grammar.txt
INFO:blib2to3.pgen2.driver:Writing grammar tables to /home/codespace/.cache/black/22.3.0/
↪Grammar3.8.13.final.0.pickle
INFO:blib2to3.pgen2.driver:Writing failed: [Errno 2] No such file or directory: '/home/
```

(continues on next page)

(continued from previous page)

```
↪codespace/.cache/black/22.3.0/tmp8s6jihtc'
INFO:blib2to3.pgen2.driver:Generating grammar tables from /workspaces/projects/venv/lib/
↪python3.8/site-packages/blib2to3/PatternGrammar.txt
INFO:blib2to3.pgen2.driver:Writing grammar tables to /home/codespace/.cache/black/22.3.0/
↪PatternGrammar3.8.13.final.0.pickle
INFO:blib2to3.pgen2.driver:Writing failed: [Errno 2] No such file or directory: '/home/
↪codespace/.cache/black/22.3.0/tmp2yj6e00b'
INFO:ploomber.dag.dag:Building DAG DAG("parametrized")
Building task 'script':   0%|          | 0/2 [00:00<?, ?it/s]INFO:ploomber.tasks.abc.
↪NotebookRunner:Starting execution: NotebookRunner: script -> File('output/nb-log.html')
INFO:papermill:Input Notebook:  /tmp/tmprwq9ox1n.ipynb
INFO:papermill:Output Notebook: /workspaces/projects/guides/logging/parametrized/output/
↪nb-log.ipynb

Executing:   0%|          | 0/6 [00:00<?, ?cell/s]INFO:papermill:Executing notebook with␣
↪kernel: python3
INFO:papermill:Executing Cell 1---------------------------------------
INFO:papermill:Ending Cell 1-------------------------------------------

Executing:  17%|        | 1/6 [00:00<00:04,  1.12cell/s]INFO:papermill:Executing Cell 2--
↪---------------------------------------
INFO:papermill:Ending Cell 2-------------------------------------------
INFO:papermill:Executing Cell 3---------------------------------------
INFO:papermill:Ending Cell 3-------------------------------------------
INFO:papermill:Executing Cell 4---------------------------------------
INFO:papermill:Ending Cell 4-------------------------------------------
INFO:papermill:Executing Cell 5---------------------------------------
INFO:papermill:Ending Cell 5-------------------------------------------
INFO:papermill:Executing Cell 6---------------------------------------
INFO:papermill:INFO:__main__:[script log] Finished step 1...

INFO:papermill:INFO:__main__:[script log] Finished step 2...

INFO:papermill:INFO:__main__:[script log] Finished step 3...

INFO:papermill:INFO:__main__:[script log] Done.

INFO:papermill:Ending Cell 6-------------------------------------------

Executing: 100%|| 6/6 [00:04<00:00,  1.44cell/s]
INFO:ploomber.tasks.abc.NotebookRunner:Done. Operation took 4.5 seconds
Building task 'function':  50%|     | 1/2 [00:04<00:04,  4.49s/it]INFO:ploomber.tasks.
↪abc.PythonCallable:Starting execution: PythonCallable: function -> File('output/fn-log.
↪txt')
INFO:tasks:[function log] Finished step 1...
INFO:tasks:[function log] Finished step 2...
INFO:tasks:[function log] Finished step 3...
INFO:tasks:[function log] Done.
INFO:ploomber.tasks.abc.PythonCallable:Done. Operation took 3.0 seconds
Building task 'function': 100%|| 2/2 [00:07<00:00,  3.76s/it]
INFO:ploomber.dag.dag: DAG report:
name      Ran?      Elapsed (s)    Percentage
```

(continues on next page)

(continued from previous page)

```
--------  ------  -------------  ------------
script    True          4.49249       59.9239
function  True          3.0045        40.0761
```

Let's now run the pipeline but switch the logging level to debug, this will print the records we saw above, plus the ones with debug level:

[5]:
```
%%sh
   parametrized
ploomber build --log debug --env--logging_level debug --force
```

```
Loading pipeline...
name      Ran?      Elapsed (s)    Percentage
--------  ------  -------------  ------------
script    True          4.47259       59.8168
function  True          3.00456       40.1832
```

```
DEBUG:ploomber.spec.dagspec:DAGSpec enviroment:
EnvDict({'cwd': '/workspaces/.../parametrized', 'git': 'master', 'git_hash': '1738307-
↪dirty', 'here': '/workspaces/.../parametrized', ...})
DEBUG:ploomber.spec.dagspec:Expanded DAGSpec:
{   'tasks': [   {   'papermill_params': {'log_output': True},
                     'params': {'logging_level': '{{logging_level}}'},
                     'product': 'output/nb-log.html',
                     'source': 'script.py'},
                 {   'params': {'logging_level': '{{logging_level}}'},
                     'product': 'output/fn-log.txt',
                     'source': 'tasks.function'}]}
WARNING:traitlets:Kernel Provisioning: The 'local-provisioner' is not found.  This is␣
↪likely due to the presence of multiple jupyter_client distributions and a previous␣
↪distribution is being used as the source for entrypoints - which does not include
↪'local-provisioner'.  That distribution should be removed such that only the version-
↪appropriate distribution remains (version >= 7).  Until then, a 'local-provisioner'␣
↪entrypoint will be automatically constructed and used.
The candidate distribution locations are: ['/workspaces/projects/venv/lib/python3.8/site-
↪packages/jupyter_client-7.3.2.dist-info']
DEBUG:ploomber.tasks.abc.NotebookRunner:Setting "script" status to TaskStatus.
↪WaitingRender
DEBUG:ploomber.tasks.abc.PythonCallable:Setting "function" status to TaskStatus.
↪WaitingRender
DEBUG:ploomber.spec.dagspec:Extracted upstream dependencies for task script: None
DEBUG:ploomber.spec.dagspec:Extracted upstream dependencies for task function: None
INFO:blib2to3.pgen2.driver:Generating grammar tables from /workspaces/projects/venv/lib/
↪python3.8/site-packages/blib2to3/Grammar.txt
INFO:blib2to3.pgen2.driver:Writing grammar tables to /home/codespace/.cache/black/22.3.0/
↪Grammar3.8.13.final.0.pickle
INFO:blib2to3.pgen2.driver:Writing failed: [Errno 2] No such file or directory: '/home/
↪codespace/.cache/black/22.3.0/tmp55q_ywf2'
INFO:blib2to3.pgen2.driver:Generating grammar tables from /workspaces/projects/venv/lib/
↪python3.8/site-packages/blib2to3/PatternGrammar.txt
INFO:blib2to3.pgen2.driver:Writing grammar tables to /home/codespace/.cache/black/22.3.0/
↪PatternGrammar3.8.13.final.0.pickle
INFO:blib2to3.pgen2.driver:Writing failed: [Errno 2] No such file or directory: '/home/
```

(continues on next page)

(continued from previous page)

```
↪codespace/.cache/black/22.3.0/tmpvxyhyfii'
INFO:ploomber.dag.dag:Building DAG DAG("parametrized")
Building task 'script':   0%|          | 0/2 [00:00<?, ?it/s]INFO:ploomber.tasks.abc.
↪NotebookRunner:Starting execution: NotebookRunner: script -> File('output/nb-log.html')
INFO:papermill:Input Notebook:  /tmp/tmpue38o2oe.ipynb
INFO:papermill:Output Notebook: /workspaces/projects/guides/logging/parametrized/output/
↪nb-log.ipynb

Executing:   0%|          | 0/6 [00:00<?, ?cell/s]INFO:papermill:Executing notebook with␣
↪kernel: python3
INFO:papermill:Executing Cell 1---------------------------------------
INFO:papermill:Ending Cell 1-----------------------------------------

Executing:  17%|        | 1/6 [00:00<00:04,  1.13cell/s]INFO:papermill:Executing Cell 2--
↪-------------------------------------
INFO:papermill:Ending Cell 2-----------------------------------------
INFO:papermill:Executing Cell 3---------------------------------------
INFO:papermill:Ending Cell 3-----------------------------------------
INFO:papermill:Executing Cell 4---------------------------------------
INFO:papermill:Ending Cell 4-----------------------------------------
INFO:papermill:Executing Cell 5---------------------------------------
INFO:papermill:Ending Cell 5-----------------------------------------
INFO:papermill:Executing Cell 6---------------------------------------
INFO:papermill:DEBUG:__main__:[script log] This is a message for debugging

INFO:papermill:INFO:__main__:[script log] Finished step 1...

INFO:papermill:INFO:__main__:[script log] Finished step 2...

INFO:papermill:INFO:__main__:[script log] Finished step 3...

INFO:papermill:INFO:__main__:[script log] Done.

INFO:papermill:Ending Cell 6-----------------------------------------

Executing: 100%|| 6/6 [00:04<00:00,  1.44cell/s]
INFO:ploomber.tasks.abc.NotebookRunner:Done. Operation took 4.5 seconds
Building task 'function':  50%|     | 1/2 [00:04<00:04,  4.47s/it]INFO:ploomber.tasks.
↪abc.PythonCallable:Starting execution: PythonCallable: function -> File('output/fn-log.
↪txt')
INFO:tasks:[function log] Finished step 1...
INFO:tasks:[function log] Finished step 2...
INFO:tasks:[function log] Finished step 3...
INFO:tasks:[function log] Done.
INFO:ploomber.tasks.abc.PythonCallable:Done. Operation took 3.0 seconds
Building task 'function': 100%|| 2/2 [00:07<00:00,  3.75s/it]
INFO:ploomber.dag.dag: DAG report:
name      Ran?      Elapsed (s)    Percentage
--------  ------  -------------  ------------
script    True          4.47259       59.8168
function  True          3.00456       40.1832
```

To get the code for the previous example, click here.

Implementation details

To keep the tutorial short, we overlooked some technical details. However, if you want to customize logging, they are essential to know.

Function tasks and sub-processes

By default, Ploomber runs function tasks in a child process. However, beginning on version 3.8, Python 3.8 switched to use spawn instead of fork on macOS, this implies that child processes *do not* inherit the logging configuration of their parents. That's why you must configure a logger inside the function's body:

```
import


def some_task
    # the following line is required on Python>=3.8 if using macOS
           .                 =       .

           =        .

    # to log a message, call logger.info
          .     'Some message'
```

Scripts and notebooks

Unlike function tasks, which can run in the same process as Ploomber or in a child process, scripts and notebooks execute independently. Hence, any logging configuration made in the main process is lost, and We have to configure a separate logger at the top of the script/notebook.

Parallel execution

Logging is currently unavailable when using the `Parallel` executor.

To run this locally, install Ploomber and execute: `ploomber examples -n guides/serialization`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.3.15 Serialization

Tutorial explaining how the serializer and unserializer fields in a pipeline.yaml file work.

Incremental builds allow Ploomber to skip tasks whose source code hasn't changed; each task must save their products to disk to enable such a feature. However, there are some cases when we don't want our pipeline to perform disk operations. For example, if we're going to deploy our pipeline, eliminating disk operations reduces runtime considerably.

To enable a pipeline to work in both disk-based and in-memory scenarios, we can declare a `serializer` and `unzerializer` in our pipeline declaration, effectively separating our task's logic from the read/write logic.

Note that this only applies to function tasks; other tasks are unaffected by the `serializer/unserializer` configuration.

Built-in pickle serialization

The easiest way to get started is to use the built-in serializer and unserializer, which use the `pickle` module.

Let's see an example; the following pipeline has two tasks. The first one generates a dictionary, and the second one creates two dictionaries. Since we are using the pickle-based serialization, each dictionary is saved in the pickle binary format:

```
# Content of simple.yaml
serializer  ploomber.io.serializer_pickle
unserializer  ploomber.io.unserializer_pickle

tasks
    source  tasks.first
    product  output/one_dict

    source  tasks.second
    product
        another  output/another_dict
        final  output/final_dict
```

Let's take a look at the task's source code:

```
# Content of tasks.py
def first
    return       =1   =2


def second
          =          'first'
            =       =      'b'  + 1   =      'a'  + 1
          =       =100   =200
    return             =             =
```

Since we configured a `serializer` and `unserializer`, function tasks must `return` their outpues instead of saving them to disk in the function's body.

`first` does not have any upstream dependencies and returns a dictionary. `second` has the previous task as dependency and returns two dictionaries. Note that the keys in the returned dictionary must match the names of the products declared in `pipeline.yaml` (`another`, `final`).

Let's now run the pipeline.

[1]:
```
%%sh
ploomber build --entry-point simple.yaml --force
```

```
name    Ran?      Elapsed (s)    Percentage
------  ------  -------------  ------------
first   True         0.001281       42.9433
second  True         0.001702       57.0567
```

```
Building task 'second': 100%|| 2/2 [00:06<00:00,  3.39s/it]
```

The pickle format has important security concerns, **remember only to unpickle data you trust**.

Custom serialization logic

We can also define our own serialization logic, by using the `@serializer`, and `@unserializer` decorators. Let's replicate what our pickle-based serializer/unserializer is doing as an example:

```
# Content of custom.py
from         import
import

from             import


@serializer
def my_pickle_serializer
                 .                  .


@unserializer
def my_pickle_unserializer
    return       .                   .
```

A `@serializer` function must take two arguments: the object to serializer and the product object (taken from the task declaration). The `@unserializer` must take a single argument (the product to unserializer), and return the unserializer object.

Let's modify our original pipeline to use this serializer/unserializer:

```
# Content of custom.yaml
serializer  custom.my_pickle_serializer
unserializer  custom.my_pickle_unserializer

tasks
    source  tasks.first
    product  output/one_dict

    source  tasks.second
    product
        another  output/another_dict
        final  output/final_dict
```

[2]:
```
%%sh
ploomber build --entry-point custom.yaml --force
```

```
name    Ran?      Elapsed (s)    Percentage
------  ------  -------------  ------------
first   True         0.001216       19.4342
second  True         0.005041       80.5658
```

```
Building task 'second': 100%|| 2/2 [00:07<00:00,  3.87s/it]
```

Custom serialization logic based on the product's extension

Under many circumstances, there are more suitable formats than pickle. For example, we may want to store lists or dictionaries as JSON files and other files using pickle. The `@serializer/@unserializer` decorators use mapping as the first argument to dispatch to different functions depending on the product's extension. Let's see an example:

```
# Content of custom.py
from         import
import
import

from             import


def write_json
                 .               .


def read_json
    return     .                 .


@serializer  '.json'
def my_serializer
                 .                 .


@unserializer  '.json'
def my_unserializer
    return       .                  .
```

Let's modify our example pipeline. The product in the first task does not have an extension (`output/one_dict`), hence, it will use pickle-based logic. However, the tasks in the second task have a `.json` extension, and will be saved as JSON files.

```
# Content of with-json.yaml
serializer  custom.my_serializer
unserializer  custom.my_unserializer

tasks
    source  tasks.first
    product  output/one_dict

    source  tasks.second
    product
        another  output/another_dict.json
        final  output/final_dict.json
```

[3]:
```
%%sh
ploomber build --entry-point with-json.yaml --force
```

```
name    Ran?      Elapsed (s)    Percentage
------  ------  -------------  ------------
first   True         0.001193       38.5834
```

(continues on next page)

(continued from previous page)

```
second  True         0.001899       61.4166
```

```
Building task 'second': 100%|| 2/2 [00:06<00:00,  3.26s/it]
```

Let's print the `.json` files to verify they're not pickle files:

[4]:
```
%%sh
cat output/another_dict.json
```

```
{"a": 3, "b": 2}
```

[5]:
```
%%sh
cat output/final_dict.json
```

```
{"a": 100, "b": 200}
```

Using a fallback format

Since it's common to have a `fallback` serialization format, the decorators have a `fallback` argument that, when enabled, uses the `pickle` module when the product's extension does not match any of the registered ones in the first argument.

The example works the same as the previous one, except we don't have to write our pickle-based logic.

`fallback` can also take the joblib or cloudpickle values. They're similar to the pickle format but have some advantages. For example, `joblib` produces smaller files when the serialized object contains many NumPy arrays, while cloudpickle supports serialization of some objects that the pickle module doesn't. To use `fallback='joblib'` or `fallback='cloudpickle'` the corresponding module must be installed.

```
# Content of custom.py

from             import


@serializer  '.json'                        =True
def my_fallback_serializer
    pass


@unserializer  '.json'                       =True
def my_fallback_unserializer
    pass
```

```
# Content of fallback.yaml
serializer  custom.my_fallback_serializer
unserializer  custom.my_fallback_unserializer

tasks
    source  tasks.first
    product  output/one_dict

    source  tasks.second
    product
```

(continues on next page)

(continued from previous page)

```
        another  output/another_dict.json
        final  output/final_dict.json
```

[6]:
```
%%sh
ploomber build --entry-point fallback.yaml --force
```

```
name    Ran?      Elapsed (s)    Percentage
------  ------  -------------  ------------
first   True         0.002278       56.8505
second  True         0.001729       43.1495
```

```
Building task 'second': 100%|| 2/2 [00:06<00:00,  3.45s/it]
```

Let's print the JSON files to verify their contents:

[7]:
```
%%sh
cat output/another_dict.json
```

```
{"a": 3, "b": 2}
```

[8]:
```
%%sh
cat output/final_dict.json
```

```
{"a": 100, "b": 200}
```

Using default serializers

Ploomber comes with a few convenient serialization functions to write more succint serializers. We can request the use of such default serializers using the `defaults` argument, which takes a list of extensions:

```
# Content of custom.py

from             import


@serializer         =True          = '.json'
def my_defaults_serializer
    pass


@unserializer         =True          = '.json'
def my_defaults_unserializer
    pass
```

Here we're asking to dispatch `.json` products and use `pickle` for all other extensions, the same as we did for the previous examples, except this time, we don't have to pass the mapping argument to the decorators.

`defaults` support:

1. `.json`: the returned object must be JSON-serializable (e.g., a list or a dictionary)
2. `.txt`: the returned object must be a string
3. `.csv`: the returned object must be a `pandas.DataFrame`

4. `.parquet`: the returned object must be a `pandas.DataFrame`, and a parquet library should be installed (such as `pyarrow`).

```
# Content of defaults.yaml
serializer  custom.my_defaults_serializer
unserializer  custom.my_defaults_unserializer

tasks
    source  tasks.first
    product  output/one_dict

    source  tasks.second
    product
        another  output/another_dict.json
        final  output/final_dict.json
```

[9]:
```
%%sh
ploomber build --entry-point defaults.yaml --force
```

```
name    Ran?      Elapsed (s)    Percentage
------  ------  -------------  ------------
first   True         0.001148       39.1675
second  True         0.001783       60.8325
```

```
Building task 'second': 100%|| 2/2 [00:07<00:00,  3.64s/it]
```

Let's print the JSON files to verify their contents:

[10]:
```
%%sh
cat output/another_dict.json
```

```
{"a": 3, "b": 2}
```

[11]:
```
%%sh
cat output/final_dict.json
```

```
{"a": 100, "b": 200}
```

Wrapping up

Configuring a `serializer` and `unserializer` in your `pipeline.yaml` is optional, but it helps you quickly generate a fully in-memory pipeline for serving predictions.

If you want to learn more about in-memory pipelines, check out the following guide.

For a complete example showing how to manage a training and a serving pipeline, and deploy it as a Flask API, click here.

2.3.16 Shell tasks

Note: Click here to see a complete example

You can run shell/bash scripts as Ploomber tasks, giving you complete flexibility to execute code in a different programming language.

Let's say you want to use a shell script to copy a file; a sample script would look like this:

```
cp origin.txt target.txt
```

To turn that into a Ploomber shell script, you need to add placeholders for upstream dependencies and the product. Let's say that `origin.txt` was generated by a task called `download` and that `target.txt` is declared as the product of the given task in the `pipeline.yaml` file, then:

```
cp {{upstream['download']}} {{product}}
```

Then, in your `pipeline.yaml`, add the task:

```
# ... more tasks

  source  copy.sh
  product  target.txt
```

Executing source code in other programming languages

By default, Ploomber will keep track of changes to your `.sh` file, so it only executes it when it changes. In our previous example, all the logic exists in the `copy.sh` file; however, if you're using shell scripts to execute source in a different programming language, say Julia, your `.sh` script may look like this:

```
julia my-script.jl input.csv output.csv
```

First, we add the placeholders:

```
julia my-script.jl {{upstream["some-task"]}} {{product}}
```

The script above will work, however, Ploomber won't track changes to `my-script.jl`. To fix this, you can use the `resources_` option, which tells Ploomber to keep track of extra files:

```
# ... more tasks

  source  scripts/run-julia.sh
  product  output.csv
  params
    resources_
        julia_script  my-script.jl
```

Note: To learn more about the `resources_` section *click here.*.

If you wish, you can update your script, to remove the duplicated `my-script.jl` value:

```
julia {{params["resources_"]["julia_script"]}} {{upstream["task"]}} {{product}}
```

2.3.17 Other editors (VSCode, PyCharm, etc.)

Note: This feature requires Ploomber `0.14` or higher.

Ploomber can be entirely operated from the command-line, thus, independent of your text editor or IDE of choice. However, Ploomber comes with a Jupyter plugin that streamlines development via the cell injection process (to learn more about cell injection, *click here*).

If you're not using Jupyter, you can still leverage the cell injection feature. Depending on your text editor/IDE capabilities, you may choose one of these options:

1. *Use the percent format in .py files* (recommended)
2. *Pair .py files with .ipynb files* (recommended if your editor does not support the `percent` format or if you're running an old `JupyterLab 1.x` version)
3. *Use .ipynb files as sources*

To try out this feature, download our `ml-basic` example:

```
ploomber examples -n templates/ml-basic -o ml-basic
```

Then move to the `ml-basic/` directory.

Using the `percent` format

Note: Editors such as **VSCode, PyCharm, Spyder, and Atom (via Hydrogen)** support the percent format.

The percent format allows you to represent `.py` files as notebooks by separating cells using `# %%`:

```
# %%
# first cell
  = 1

# %%
# second cell
  = 2
```

The first step is to ensure that your scripts are in the percent format. You can re-format all of them with the following command:

```
ploomber nb --format py:percent
```

If you're following this using our `ml-basic` example, you can run such command, then open `fit.py` and see that the cells are delimited by `# %%`.

Now, let's inject the cell into each script manually:

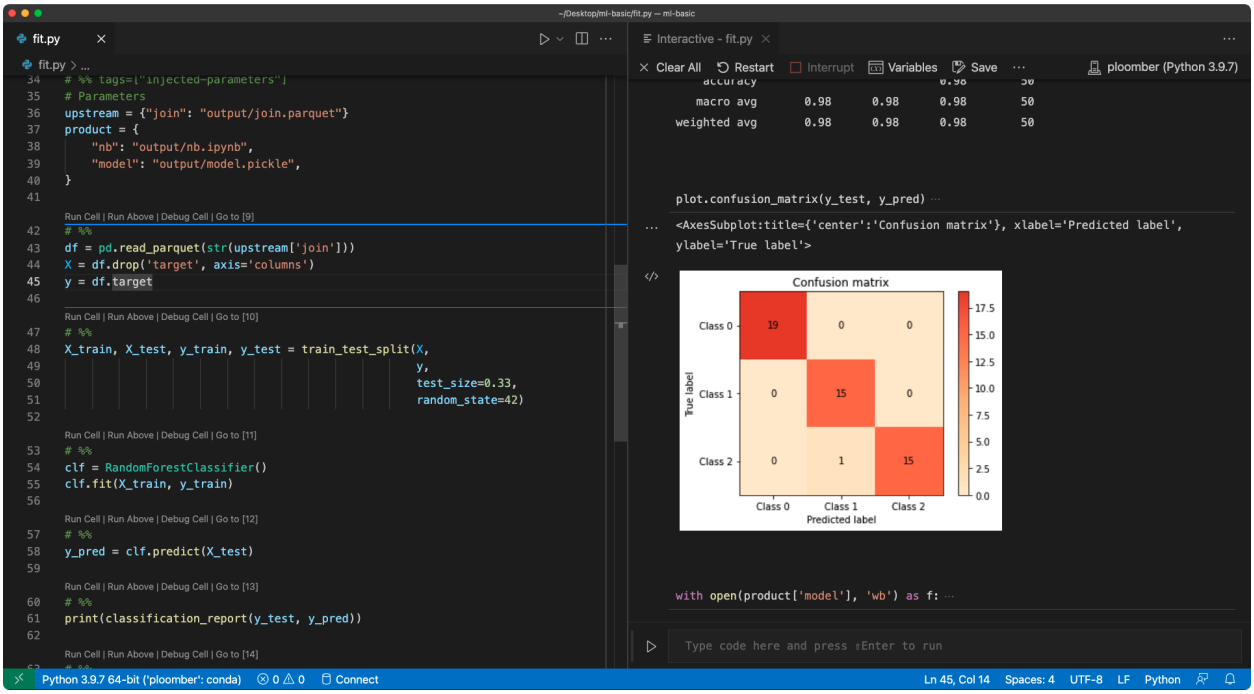
```
ploomber nb --inject
```

If you open any of your pipeline scripts, you'll see the injected cell. If you're following this with our `ml-basic` example; you'll notice that after running `ploomber nb --inject`, the `fit.py` file has a few new lines:

```
# %% tags=["injected-parameters"]
         =  "join"  "output/join.parquet"


        =
    "nb"  "output/nb.ipynb"
    "model"  "output/model.pickle"
```

In our `ml-basic` example, `fit.py` depends on the `join` task, which implies that `fit.py` will use the output of `join` as input. Once you inject the cell, you'll see that Ploomber extracted the outputs of `join` and added them to `fit.py`, now `fit.py` is complete, and you can run it interactively without hardcoding paths.

To test this, run `ploomber build` in a terminal to generate all the outputs, then open `fit.py` and start running the script (if you're on VSCode, you can click on the `Run Cell` button at the beginning of each cell). The following image shows the `fit.py` (left) and the interactive output (right) on VSCode:



Important: Remember to run `ploomber nb --inject` whenever you change your `pipeline.yaml`. You can set up a file watcher for `pipeline.yaml`. Click here for a VSCode extension, or here for a PyCharm example.

Note: By default, Ploomber deletes the injected cell when you save a script/notebook from Jupyter; however, if you injected it via the `ploomber nb --inject` command, this is disabled, and saving the script/notebook will not remove the injected cell.

Pairing `.ipynb` files

If your editor does not support the percent format, or if running and old `JupyterLab 1.x` version (e.g., if using **Amazon Sagemaker**), you can pair `.py` and `.ipynb` files: this creates a synced `.ipynb` copy of each `.py` task.

Say you have a pipeline with `.py` files, to create the `.ipynb` ones:

```
ploomber nb --pair notebooks
```

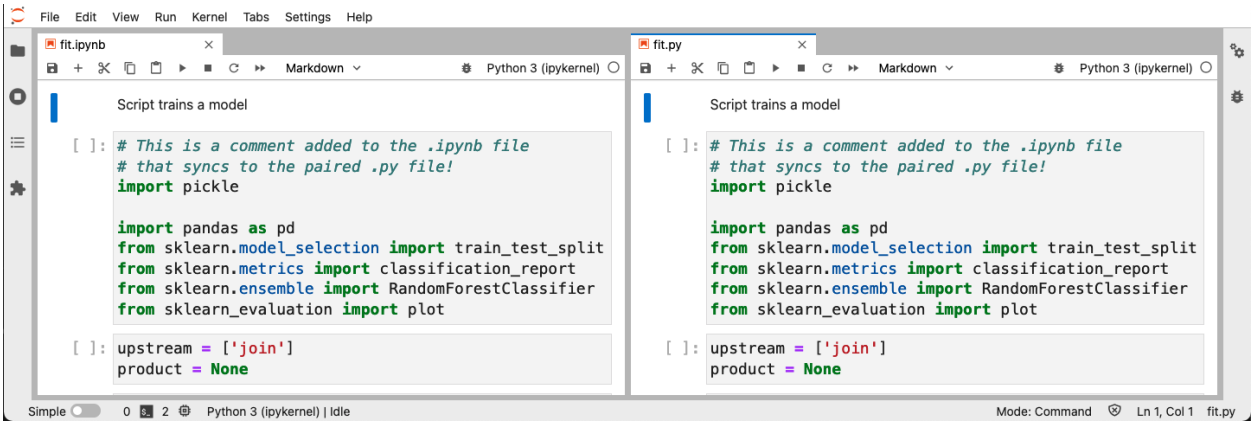
The command above will generate `.ipynb` files in a `notebooks/` directory, one per `.py` in your pipeline. If you're following the `ml-basic` example, you'll see that a new `notebooks/fit.ipynb` file will appear after running the previous command. Now, add the injected cell: `ploomber nb --inject` (more details in the the *previous section*).

Tip: Keep your repository clean by adding the `.ipynb` files to your `.gitignore` file.

Once you modify the `.ipynb`, you can sync their `.py` pairs with:

```
ploomber nb --sync
```

The following image shows the `.ipynb` / `.py` pair after running the sync command:



If you're following this using the `ml-basic` command, modify `notebooks/fit.ipynb` (e.g., add a comment in the first cell), run `ploomber nb --sync`, and then open `fit.py`, you'll see that the change made to the `.ipynb` file is now visible in the `fit.py` file.

Tip: If you want the `ploomber nb --sync` command to run automatically before you run `git push`, check out the *git hooks* section.

Using `.ipynb` as sources

As a last option, you have the option to use `.ipynb` files as task sources in your `pipeline.yaml`:

```
tasks
    source  nbs/load.ipynb
    product  output/report.ipynb
```

Keep in mind that `.ipynb` files are hard to manage with git, so we recommend you to use one of the alternative options described above.

To add the injected cell, follow the instructions from the *previous section*.

Removing the injected cell

If you wish to remove the injected cell from all scripts/notebooks:

```
ploomber nb --remove
```

Using `git` hooks

Important: `ploomber nb --install-hook` does not work on Windows

To keep your scripts/notebooks clean, it's a good idea to keep the injected cell out of version control.

To automate injecting/removing, you can install git hooks that automatically remove the injected cells before committing files and inject them again after committing:

```
ploomber nb --install-hook
```

To uninstall the hooks:

```
ploomber nb --uninstall-hook
```

2.3.18 R support

Ploomber officially supports R. The same concepts that apply to Python scripts apply to R scripts; this implies that R scripts can render as notebooks in Jupyter and the cell injection works. The only difference is how to declare `upstream` dependencies:

For the R Markdown format (`.Rmd`):

````
```{r, tags=c("parameters")}
upstream = list('one_task', 'another_task')
```
````

If you prefer, you can also use plain R scripts:

```
# %% tags=["parameters"]
        = list 'one_task'  'another_task'
#
```

If your script doesn't have dependencies: `upstream = NULL`

To read more about how Ploomber executes scripts and integrates with Jupyter, check the *Jupyter Integration guide*.

Configuring R environment

To run R scripts as Jupyter notebooks, you need to install Jupyter first, have an existing R installation and install the IRkernel package.

If you are using `conda` and a `environment.yml` file to manage dependencies, keep on reading. Otherwise, read the IRkernel installation instructions.

Setting up R and IRkernel via `conda`

Even if you already have R installed, it is good to isolate your environments from one project to another. `conda` can install R inside your project's environment.

Add the following lines to your `environment.yaml`:

```
name  some_project

dependencies
  # ...
  # existing conda dependencies...
    r-base
    r-irkernel
  # optionally add r-essentials to install commonly used R packages

    pip
    # ...
    # existing pip dependencies...
      ploomber
```

For more information on installing R via `conda` click here.

Once you update your `environment.yml`, re-create or update your environment.

Finally, activate the R kernel for Jupyter. If you're using Linux or macOS:

```
echo "IRkernel::installspec()" | Rscript -
```

If using Windows, start an R session and run `IRkernel::installspec()` on it.

Interactive example

Click the button above to see an interactive example (no installation needed, but takes about a minute to be ready):

Example source code

2.3.19 FAQ and Glossary

Why do products have clients?

Clients exist in tasks and products because they serve different purposes. A task client manages the connection to the database that runs your script. On the other hand, the product's client only handles the storage of the product's metadata.

To enable incremental runs. Ploomber has to store the source code that generated any given product. Storing metadata in the same database that runs your code requires a system-specific implementation. Currently, only SQLite and PostgreSQL are supported via *ploomber.products.SQLiteRelation* and *ploomber.products.PostgresRelation* respectively. For these two cases, task client and product client communicate to the same system (the database). Hence they can initialize with the same client.

For any other database, we provide two alternatives; in both cases, the task's client is different from the product's client. The first alternative is *ploomber.products.GenericSQLRelation* which represents a generic table or view and saves metadata in a SQLite database; on this case, the task's client is the database client (e.g., Oracle, Hive, Snowflake) but the product's client is a SQLite client. If you don't need the incremental builds features, you can use *ploomber.products.SQLRelation* instead which is a product with no metadata.

Which databases are supported?

The answer depends on the task to use. There are two types of database clients. *ploomber.clients.SQLAlchemyClient* for `SQLAlchemy` compatible database and *ploomber.clients.DBAPIClient* for the rest (the only requirement for `DBAPIClient` is a driver that implements PEP 249.

ploomber.tasks.SQLDump supports both types of clients.

ploomber.tasks.SQLScript supports both types of clients. But if you want incremental builds, you must also configure a product client. See the section below for details.

ploomber.tasks.SQLUpload relies on *pandas.to_sql* to upload a local file to a database. Such method relies on SQLAlchemy to work. Hence it only supports `SQLAlchemyClient`.

ploomber.tasks.PostgresCopyFrom is a faster alternative to `SQLUpload` when using PostgreSQL. It relies on *pandas.to_sql* **only** to create the database, but actual data upload is done using `psycopg` which calls the native `COPY FROM` procedure.

What are incremental builds?

When developing pipelines, we usually make small changes and want to see how the the final output looks like (e.g., add a feature to a model training pipeline). Incremental builds allow us to skip redundant work by only executing tasks whose source code has changed since the last execution. To do so, Ploomber has to save the Product's metadata. For *ploomber.products.File*, it creates another file in the same location, for SQL products such as *ploomber.products.SQLRelation*, a metadata backend is required, which is configured using the `client` parameter.

How do I specify a task with a variable number of outputs?

You must group the outputs into a single product and declare it as a directory.

- Click here to see an example.
- If you're using serializers, click here to see an example.

Should tasks generate products?

Yes. Tasks must generate at least one product; this is typically a file but can be a table or view in a database.

If you find yourself trying to write a task that generates no outputs, consider the following options:

1. Merge the code that does not generate outputs with upstream tasks that generate outputs.
2. Use the `on_finish` hook to execute code after a task executes successfully (click *here* to learn more).

Auto reloading code in Jupyter

When you import a module in Python (e.g., `from module import my_function`), the system caches the code and subsequent changes to `my_funcion` won't take effect even if you run the import statement again until you restar the kernel, which is inconvenient if you are iterating on some code stored in an external file.

To overcome such limitation, you can insert the following at the top of your notebook, before any `import` statements:

```
# auto reload modules
%
%           2
```

Once executed, any updates to imported modules will take effect if you change the source code. Note that this feature has some limitations.

Cell tags

Scripts (with the `%%` separator) and notebooks support cell tags. Tags help identify cells for several purposes; the two most common ones are:

1. Notebook parameters: Ploomber uses this cell to know where to inject parameters from `pipeline.yaml`. In most cases, you don't need to manually tag cells since Ploomber does it automatically.
2. Cell filtering: When generating reports, you may want to selectively hide specific cells from the output report.

For an example of adding cell tags, see the *Parameterizing Notebooks* section.

Parameterizing Notebooks

You must first parametrize the notebook by assigning the tag `parameters` to an initial cell when performing a notebook task. Note that the parameters in the `parameters` cell are placeholders; they indicate the parameter names that your script or notebook takes, but they are replaced values declared in your `pipeline.yaml` file at runtime. The only exception is the `upstream` parameter, which contains a list of task dependencies.

Parameterizing `.py` files

For `.py` files, include the `# %% tags=["parameters"]` comment before declaring your default variables or parameters.

```
# %% tags=["parameters"]
upstream = None
product = None
```

Note that Ploomber is compatible with all `.py` formats supported by jupytext. Another common alternative is the light format. The `# +` marker denotes the beginning of a cell, and `# -` marker indicates the end of the cell. Your cell should look like this:

```
# + tags=["parameters"]
upstream = None
product = None
# -
```

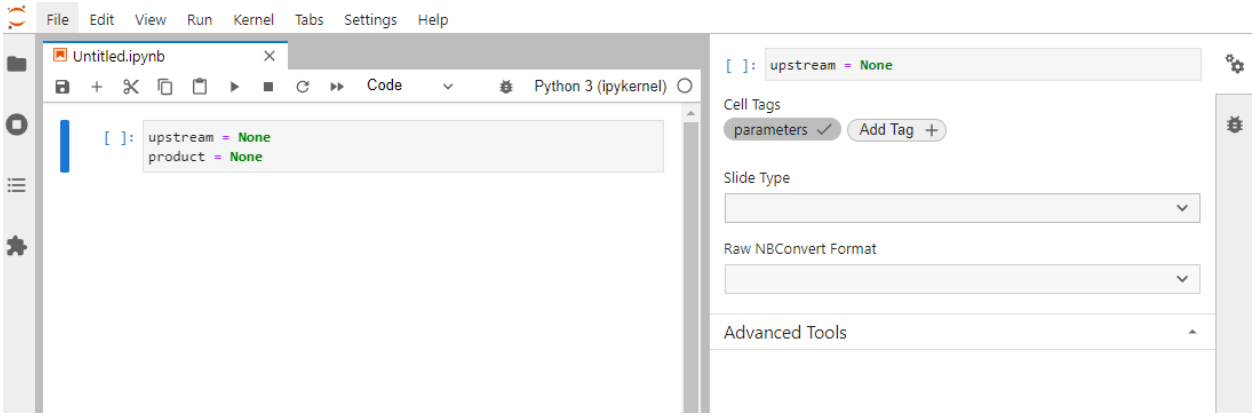
If you're using another format, check out jupytext's documentation.

Parameterizing `.ipynb` files in Jupyter

Note: This applies to JupyterLab 3.0 and higher. For more information on parameterizing notebooks in older versions, please refer to papermill docs

---

To parametrize your notebooks, add a new cell at the top, then in the right sidebar, click to open the property inspector (double gear icon). Next, hit the "Add Tag" button, type in the word `parameters`, and press "Enter".



Plotting a pipeline

You can generate a plot of your pipeline with `ploomber plot`. It supports using D3, mermaid.js and `pygraphviz` as backends to create the plot. D3 is the most straightforward option since it doesn't require any extra dependencies, but `pygraphviz` is more flexible and produces a better plot. Once installed, Ploomber will use `pygraphviz`, but you can use the `--backend` argument in the `ploomber plot` command to switch between `d3`, `mermaid`, and `pygraphviz`.

The simplest way to install `pygraphviz` is to use `conda`, but you can also get it working with `pip`.

`conda` (simplest)

```
conda install pygraphviz -c conda-forge
```

Important: If you're running Python `3.7.x`, run: `conda install 'pygraphviz<1.8' -c conda-forge`

`pip`

`graphviz` cannot be installed via `pip`, so you must install it with another package manager, if you have `brew`, you can get it with:

```
brew install graphviz
```

Note: If you don't have `brew`, refer to graphviz docs for alternatives.

Once you have `graphviz`, you can install `pygraphviz` with `pip`:

```
pip install pygraphviz
```

Important: If you're running Python `3.7.x`, run: `pip install 'pygraphviz<1.8'`

Can I use Ploomber in old JupyterLab `1.x` versions?

Yes! Although our JupyterLab plug-in requires version `2.x`, you can still use Ploomber if using the old `1.x` version, which (as of December 2021) is the case if you're using **Amazon Sagemaker**. Since Ploomber is a command-line tool, it is independent of your editor/IDE. Furthermore, you can get the same experience as JupyerLab users by using the `ploomber nb` command; *click here to learn more.*

Multiprocessing errors on macOS and Windows

Show me the solution.

By default, Ploomber executes *ploomber.tasks.PythonCallable* (i.e., function tasks) in a child process using the multiprocessing library. On macOS and Windows, Python uses the spawn method to create child processes; this isn't an issue if you're running your pipeline from the command-line (i.e., `ploomber build`), but you'll encounter the following issue if running from a script:

```
An attempt has been made to start a new process before the current process
has finished its bootstrapping phase.

This probably means that you are not using fork to start your
child processes and you have forgotten to use the proper idiom
in the main module:

    if __name__ == '__main__':
        freeze_support()
        ...

The "freeze_support()" line can be omitted if the program
is not going to be frozen to produce an executable.
```

This happens if you store a script (say `run.py`):

```
from               import

    =         'pipeline.yaml' .
# This fails on macOS and Windows!
   .
```

And call your pipeline with:

```
python run.py
```

There are two ways to solve this problem.

Solution 1: Add `__name__ == '__main__'`

To allow correct creation of child processes using `spawn`, run your pipeline like this:

```
from               import

if          == '__main__'
        =         'pipeline.yaml' .
    # calling build under this if statement allows
    # correct creation of child processes
       .
```

Solution 2: Disable multiprocessing

You can disable multiprocessing in your pipeline like this:

```
from               import
from                   import

    =          'pipeline.yaml' .
# overwrite executor regardless of what the pipeline.yaml
# says in the 'executor' field
   .         =                            =False


   .
```

Glossary

1. **Dotted path**. A dot-separated string pointing to a Python module/class/function, e.g. "my_module.my_function".
2. **Entry point**. A location to tell Ploomber how to initialize a DAG, can be a spec file, a directory, or a dotted path
3. **Hook**. A function executed after a certain event happens, e.g., the task "on finish" hook executes after the task executes successfully
4. **Spec**. A dictionary-like specification to initialize a DAG, usually provided via a YAML file

2.3.20 Spec API vs. Python API

There are two ways of writing pipelines with Ploomber. This document discusses the differences and how to decide which API to use.

Data projects span a wide range of applications, from small projects requiring a few scripts to large ones requiring greater flexibility.

If you're getting started, the Spec API is recommended. The Spec API is flexible enough to handle many common use cases without requiring you to learn a Python API, you can get started quickly and get pretty far using some of the advanced feature, check out the full *documentation* for details.

However, the Spec API is static, meaning that your `pipeline.yaml` completely describes your pipeline's structure. Under some circumstances, you may want your pipeline to be more "dynamic." For example, you may use some input parameters and create a "pipeline factory," which is a function that takes those input parameters and creates a pipeline, allowing you to morph the specifics of your pipeline (number of tasks, dependencies among them, etc.) dynamically, you can only achieve so via the Python API is a. The downside is that it has a steeper learning curve.

There is a third way of assembling a pipeline by pointing to a *directory* with scripts. This API allows you quickly test simple pipelines that may only have a couple of tasks.

For examples using different APIs, click here

Depending on the API you use, the pipeline will be exposed to Ploomber differently. For example, if using the Spec API, you tell the pipeline to Ploomber by pointing to the path to a `pipeline.yaml` file, known as an entry point, discussed below.

Entry points

To execute your pipeline, Ploomber needs to know where it is. This location is known as "entry point". There are three types of entry points (ordered by flexibility):

1. A directory
2. [Spec API] Spec (aka `pipeline.yaml`)
3. [Python API] Factory function (a function that returns a `DAG` object)

The following sections describe each entry point in detail.

Directory entry point

Ploomber can figure out your pipeline without even having a `pipeline.yaml`, by just passing a directory. This kind of entry point is the simplest one but also, the less flexible, since there isn't a pipeline definition, **products must be declared in the source code itself**. Note that only Python and R scripts support this, for example:

```
# %% tags=["parameters"]
        =  'nb'  'output.ipynb'  'data'  'output.csv'
         =  'a_task'
#

# continues...
```

Internally, Ploomber uses the *ploomber.spec.DAGSpec.from_directory* method. See the documentation for details.

All commands that accept the `--entry-point/-e` parameter can take a directory as a value. For example, to build a pipeline using the current directory:

```
ploomber build --entry-point .
```

It's also possible to select a subset of the files in a directory using a glob-like pattern:

```
ploomber build --entry-point "*.py" # note the quotes
```

Note: Pipelines built without a `pipeline.yaml` file cannot be parametrized.

[Spec API] Spec entry point

If you want to customize how Ploomber executes your pipeline, you have to create a `pipeline.yaml` file; this is known as a **spec entry point**. A `pipeline.yaml` file is the recommended approach for most projects: it has a good level of flexibility and doesn't require you to learn Ploomber's internal Python API.

To call a DAG defined in a `path/to/pipeline.yaml` file pass the path:

```
ploomber build --entry-point pah/to/pipeline.yaml
```

If your pipeline exists inside a package:

```
ploomber build --entry-point my_package::pah/to/pipeline.yaml
```

The command above searches for package `my_package` (by doing `import my_package`), then uses the relative path.

You can omit the `--entry-point` argument if the `pipeline.yaml` is in a standard location (*Default locations*).

An added feature is pipeline parametrization, to learn more *Parametrized pipelines*.

For schema details see: *Spec API (pipeline.yaml)*.

[Python API] Factory entry point

The last approach requires you to write Python code to specify your pipeline. It has a steeper learning curve because you have to become familiar with the API specifics, but it provides the most significant level of flexibility.

The primary advantage is dynamic pipelines, whose exact number of tasks and dependency relations are determined when executing your Python code. For example, you might use a for loop to dynamically generate a few tasks based on some input parameters.

For Ploomber to know how to build your pipeline written as Python code, you have to provide a **factory entry point**, which is a function that returns a `DAG` object. For example, if your factory is a function called *make* in a file called `pipeline.py`, then your entry point is the dotted path `pipeline.make`, which may look like this:

```
from          import

def make
        =
    # add tasks to your pipeline...
    return
```

You can execute commands against your pipeline like this:

```
ploomber {command} --entry-point pipeline.make
```

Internally, Ploomber will do something like this:

```
from          import

    =

# (if using ploomber build)
   .
```

If your factory function has arguments, they will show up in the CLI. This guide shows how to parametrize a factory function: *Parametrized pipelines*

If your factory function has a docstring, the first line displays in the CLI help menu (e.g. `ploomber build --entry-point factory.make --help`). If the docstring is in the numpydoc format (and numpydoc is installed, `pip install numpydoc`), descriptions for documented parameters will be displayed as well.

2.4 Deployment

2.4.1 Introduction

The two most common ways to deploy data pipelines are batch and online. Ploomber supports both deployment options.

In batch, you obtain new data to make predictions and store them for later use. This process usually happens on a schedule. For example, you may develop a Machine Learning pipeline that runs every morning, predicts the probability of user churn, and stores such probabilities in a database table.

Alternatively, you may deploy a pipeline as an online service and expose your model as a REST API; users request predictions at any time by sending input data.

Pipeline composition

Before diving into deployment details, let's introduce the concept of pipeline composition.

The only difference between a Machine Learning training pipeline and its serving counterpart is what happens at the beginning and the end.

At **training** time, we obtain historical data, generate features, and train a model:

At **serving** time, we obtain new data, generate features and make predictions using a trained model:

When the feature engineering process does not match, training-serving skew arises. Training-serving skew is one of the most common problems when deploying ML models. To fix it, Ploomber allows you to compose pipelines: **write your feature generation once and re-use it to organize your training and serving pipelines**; this ensures that the feature engineering code matches precisely.

2.4.2 Batch processing

You can export Ploomber pipelines to production schedulers for batch processing. Check out our package Soopervisor, which allows you to export to Kubernetes (via Argo workflows), AWS Batch, Airflow, and SLURM.

Composing batch pipelines

To compose a batch pipeline, use the `import_tasks_from` directive in your `pipeline.yaml` file.

For example, define your feature generation tasks in a `features.yaml` file:

```
# generate one feature...
  source  features.a_feature
  product  features/a_feature.csv

# another feature...
  source  features.anoter_feature
  product  features/another_feature.csv

# join the two previous features...
  source  features.join
  product  features/all.csv
```

Then import those tasks in your training pipeline, `pipeline.yaml`:

```
meta
    # import feature generation tasks
    import_tasks_from  features.yaml

tasks
    # Get raw data for training
      source  train.get_historical_data
      product  raw/get.csv

    # The import_tasks_from injects your features generation tasks here
```

(continues on next page)

(continued from previous page)

```
    # Train a model
      source  train.train_model
      product  model/model.pickle
```

Your serving pipeline `pipepline-serve.yaml` would look like this:

```
meta
    # import feature generation tasks
    import_tasks_from  features.yaml

tasks
    # Get new data for predictions
      source  serve.get_new_data
      product  serve/get.parquet

    # The import_tasks_from injects your features generation tasks here

    # Make predictions using a trained model
      source  serve.predict
      product  serve/predictions.csv
      params
        path_to_model  model.pickle
```

Here's an example project showing how to use `import_tasks_from` to create a training (`pipeline.yaml`) and serving (`pipeline-serve.yaml`) pipeline.

Scheduling

For an example showing how to schedule runs with cron and Ploomber, click here.

2.4.3 Online service (API)

To encapsulate all your pipeline's logic for online predictions, use *ploomber.OnlineDAG*. Once implemented, you can generate predictions like this:

```
from            import

# MyOnlineDAG is a subclass of OnlineDAG
    =
   .                =
```

You can easily integrate an online DAG with any library such as Flask or gRPC.

The only requisite is that your feature generation code should be entirely made of Python functions (i.e., *ploomber.tasks.PythonCallable*) tasks with configured *serializer and unserializer*.

Composing online pipelines

To create an online DAG, list your feature tasks in a `features.yaml` and use `import_tasks_from` in your training pipeline (`pipeline.yaml`). Subclass *ploomber.OnlineDAG* to create a serving pipeline.

`OnlineDAG` will take your tasks from `features.yaml` and create new "input tasks" based on `upstream` references in yout feature tasks.

For example, if `features.yaml` has tasks `a_feature` and `another_feature` (see the diagram in the first section), and both obtain their inputs from a task named `get`; the source code may look like this:

```
def a_feature
            =          'get'
    # process raw_data to generate features...
    # return a_feature
    return

def another_feature
            =          'get'
    # process raw_data to generate features...
    # return another_feature
    return
```

Since `features.yaml` does not contain a task named `get`, `OnlineDAG` automatically identifies it as an "input task". Finally, you must provide a "terminal task", which is the last task in your online pipeline:

To implement this, create a subclass of `OnlineDAG` and provide the path to your `features.yaml`, parameters for your terminal task and the terminal task:

```
from          import

# subclass OnlineDAG...
class MyOnlineDAG
    # and provide these three methods...

    # get_partial: returns a path to your feature tasks
    @staticmethod
    def get_partial
        return 'tasks-features.yaml'

    # terminal_params: returns a dictionary with parameters for the terminal task
    @staticmethod
    def terminal_params
             =       .              .                        'model.pickle'
        return           =

    # terminal_task: implementation of your terminal task
    @staticmethod
    def terminal_task
        # receives all tasks with no downtream dependencies in
        # tasks-features.yaml
                 =          'a_feature'
                       =          'another_feature'
          =    .           'a_feature'
                           'anoter_feature'
```

(continues on next page)

(continued from previous page)

```
        return      .
```

To call `MyOnlineDAG`:

```
from            import

    =

# pass parameters (one per input)
           =    .           =
```

You can import and call `MyOnlineDAG` in any framework (e.g., Flask) to expose your pipeline as an online service.

```
from       import
import        as

from            import

# instantiate online dag
    =
    =

@app.      '/'         = 'POST'
def predict
                =        .
    # get JSON data and create a data frame with a single row
              =   .                                = 0
    # pass input data, argument per root node
        =         .           =
    # return output from the terminal task
    return          'prediction'          'terminal'
```

Examples

Click here to see a deployment example using AWS Lambda.

Click here to see a complete sample project that trains a model and exposes an API via Flask.

2.4.4 Large-scale training

Ploomber pipelines can export to distributed workflow orchestrators for large-scale training. Check out our package Soopervisor, which allows you to export to:

1. Kubernetes (via Argo workflows)
2. AWS Batch
3. Airflow
4. SLURM

2.4.5 Custom deployment

If don't use one of the supported platforms (Kubernetes, AWS Batch, Airflow, and SLURM.), you can deploy using the Python API.

Every Ploomber pipeline is represented as a *ploomber.DAG* object, which contains all the information you need to orchestrate the pipeline in any platform, that's how we export to other platforms: we load the user's pipeline as a DAG object and then convert it.

If you need help or have questions, open an issue or send us a message on Slack.

2.5 Cookbook

2.5.1 Parametrization

Note: This is a quick reference, for an in-depth tutorial, *click here*.

To parametrize your pipeline, create an `env.yaml`:

```
some_param  some_value
another_param  42
```

Then use `{{placeholders}}` in your `pipeline.yaml` file:

```
tasks
    source  scripts/plot.py
    product  products/plot.ipynb
    params
      some_param  '{{some_param}}'
```

When executing your pipeline, `scripts/plot.py` receives `some_param="some_value"`.

You can use `{{placeholders}}`. The most common use case are `tasks[*].params` (just like the example above), and `tasks[*].product`, to change the output location, or you can use both at the same time:

```
tasks
    source  scripts/plot.py
    product  'products/{{some_param}}/plot.ipynb'
    params
      some_param  '{{some_param}}'
```

Switching from the command line

Ploomber recognizes `{{placeholders}}` and adds command-line arguments to change their value, to see a list of available placeholders:

```
ploomber build --help
```

Dynamic parameters

Parameters declared in `env.yaml` are static (they can only change in value by editing the `env.yaml` file or via the command-line); however, you can use the Python API to create dynamic parameters whose values are determined at runtime, check out this example.

To run this locally, install Ploomber and execute: `ploomber examples -n cookbook/report-generation`

Found an issue? Let us know.

Questions? Ask us on Slack.

2.5.2 Report generation

Generating HTML/PDF reports.

Ploomber makes it simple to generate HTML and PDF reports from notebooks and scripts. To see some examples, go to the `reports/` directory. This cookbook covers several use cases and includes runnable examples.

HTML reports (easiest option)

HTML reports are the simplest option as they don't require any extra dependencies. You only need to change the `product` extension to `.html` and Ploomber will do the conversion:

```
# Content of pipeline.yaml
    # scripts can generate reports
      source  tasks/script.py
      name  html-report
      product
        nb  reports/report.html
        # the task can generate more outputs, list them here
```

Runnable example:

```
# get example
pip install ploomber "nbconvert[webpdf]" --upgrade
ploomber examples -n cookbook/report-generation -o example
   example

# install example dependencies
ploomber install

# generate HTML report
ploomber task html-report
```

Check out report at `reports/report.html`

PDF reports

To generate PDF reports there are two options, using chromium or TeX.

Using chromium (easiest pdf option)

To use use chromium, pass `nbconvert_exporter_name: webpdf`

```
# Content of pipeline.yaml
    # pdf report example
      source  tasks/script.py
      name  webpdf-report
      # use the webpdf exporter (supportes embedded charts)
      # (it will download chromium if needed)
      nbconvert_exporter_name  webpdf
      product
        nb  reports/report-webpdf.pdf
```

Runnable example:

```
# get example
pip install ploomber "nbconvert[webpdf]" --upgrade
ploomber examples -n cookbook/report-generation -o example
   example

# install example dependencies
ploomber install

# generate PDF report
ploomber task webpdf-report
```

Check out report at `reports/report-webpdf.pdf`

Using TeX

TeX is the default, to use it, set the product extension to `.pdf`:

```
# Content of pipeline.yaml
    # pdf report example (requires latex)
      source  tasks/script.py
      name  pdf-report
      # generate pdf report by changing the extension.
      product
        nb  reports/report.pdf
```

Runnable example:

```
# get example
pip install ploomber "nbconvert[webpdf]" --upgrade
ploomber examples -n cookbook/report-generation -o example
   example
```

(continues on next page)

(continued from previous page)

```
# install example dependencies
ploomber install

# generate PDF report
ploomber task pdf-report
```

Check out report at `reports/report.pdf`

Installing TeX

For instructions on installing TeX, see this..

TeXLive is a large distribution, as an alternative, you may install BasicTeX. Here are instructions for macOS.

Upon BasicTeX installation, you'll need to install a few extra packages:

```
# Note: if using macOS or Linux, you may need to execute with sudo
tlmgr install adjustbox \
  caption \
  collectbox \
  enumitem \
  environ \
  eurosym \
  jknapltx \
  parskip \
  pgf \
  rsfs \
  tcolorbox \
  titling \
  trimspaces \
  ucs \
  ulem \
  upquote
```

Source.

Hiding code

In many cases, you want to hide the code so the report only contains tables and charts, you can do so easily with the `exclude_input` option:

```
# Content of pipeline.yaml
    # notebooks as well
      source  tasks/notebook.ipynb
      name  another-html-report
      product
        nb  reports/another.html

      nbconvert_export_kwargs
        # optionally hide the code from the report
        exclude_input  True
```

Runnable example:

```
# get example
pip install ploomber "nbconvert[webpdf]" --upgrade
ploomber examples -n cookbook/report-generation -o example
   example

# install example dependencies
ploomber install

# generate HTML report and hide code
ploomber task another-html-report
```

Check out report at `reports/another.html`

Hiding cells

You may want to hide cells from the output notebook selectively. You can do so with the `TagRemovePreprocessor`, which takes a list of tags. Any cells with such tags are excluded:

```
# Content of pipeline.yaml
    # notebooks as well
      source  tasks/notebook.ipynb
      name  another-html-report
      product
        nb  reports/another.html

      nbconvert_export_kwargs
        # optionally hide the code from the report
        exclude_input  True

        # optionally, exclude cells with certain tags
        config
          HTMLExporter
            preprocessors
          TagRemovePreprocessor
            remove_cell_tags
```

To learn how to add cell tags, see this.

Runnable example:

```
# get example
pip install ploomber "nbconvert[webpdf]" --upgrade
ploomber examples -n cookbook/report-generation -o example
   example

# install example dependencies
ploomber install

# generate HTML report and hide boxplot
ploomber task another-html-report
```

Check out report at `reports/another.html`

2.5.3 Debugging

Note: This is a quick reference, for an in-depth tutorial, *click here*.

Note: All this section assumes you're familiar with the Python debugger. See the documentation here

Executing task in debugging mode

To jump to the first line of a task and start a debugging session:

```
ploomber interact
```

Then:

```
    'task-name' .
```

Note: `.debug()` only works with Python functions, scripts, and notebooks.

To get the list of task names: `list(dag)`.

After running `.debug()`, you'll start a debugging session. You can use the `next` command to jump to the next line. Type `quit`, and hit enter to exit the debugging session.

Post-mortem debugging

Run and start a debugging session as soon as a task raises an exception.

```
ploomber task {task-name} --debug
```

```
ploomber build --debug
```

New in version 0.20: Added support for post-mortem debugging in notebooks using `--debug`

New in version 0.20: Added `--debug` option to `ploomber task`

Post-mortem debugging (debug later)

Added in version 0.20

Run the pipeline and serialize errors from all failing tasks for later debugging. This is useful when running tasks in parallel or notebooks overnight.

```
ploomber task {task-name} --debuglater
```

```
ploomber build --debuglater
```

Then, to start a debugging session:

```
dltr {task-name}.dump
```

Once you're done debugging, you can delete the `{task-name}.dump` file.

Note: Only built-in objects will be stored in the `.dump` file, for others, (e.g., pandas data frames, numpy arrays) only the string representation is stored. To serialize all: `pip install 'debuglater[all]'`

Important: Using `--debuglater` will serialize all the variables, ensure you have enough disk space, especially if running tasks in parallel.

New in version 0.20: Added `--debuglater` option to `ploomber task`

New in version 0.20: Added `--debuglater` option to `ploomber build`

Breakpoints

Note: This only work with Python functions, go to the *next section* to learn how to debug scripts/notebooks.

Breakpoints allow you to start a debugger at given line:

```
def my_task
    # debugging session starts here...
    from      import
    # code continues...
```

Then:

```
ploomber build --debug
```

Debugging in Jupyter/VSCode

If you're using Jupyter or similar (e.g. notebooks in VSCode), you can debug there.

Post-portem

If your code raises an exception, execute the following in a new cell, and a debugging session will start:

```
%
```

Click here to see the `%pdb` documentation.

If you want a debugging session to start whenever your code raises an exception:

```
%
```

Note: run `%pdb` again to turn it off.

Click here to see the `%pdb` documentation.

Breakpoints

Once you're in Jupyter, you can add a breakpoint at the line you want to debug:

```
def some_code_called_from_the_notebook
    # debugging session starts here...
    from      import
    # code continues...
```

The breakpoint can be in a module (i.e., something that you imported using a `import` statement)

Visual debugger

JupyterLab recently incorporated a native debugger, click here to learn more.

2.5.4 Logging

Note: This is a quick reference, for an in-depth tutorial, *click here*.

Function tasks

If you're using functions as tasks, configure logging like this:

```
import

def some_task
    # uncomment the next line if using Python >= 3.8 on macOS
    # logging.basicConfig(level=logging.INFO)

          =       .

    # to log a message, call logger.info
         .     'Some message'
```

Scripts or notebooks

If using scripts/notebooks tasks, add this a the top of **each** one:

```
import
import

      .               =       .           =   .
     =       .

# to log a message, call logger.info
     .     'Some message'
```

and add the following to each task definition:

```
tasks
    source  scripts/script.py
    product  products/output.ipynb
    # add this
    papermill_params
      log_output  True
```

Then, use the `--log` option when building the pipeline to print records to the terminal:

```
ploomber build --log info
```

If you also want to send logs to a file:

```
ploomber build --log info --log-file my.log
```

2.5.5 Serialization

Note: This is a quick reference, for an in-depth tutorial, *click here*.

By default, tasks receive a `product` argument and must take care of serializing their outputs at the passed location. Serialization allows tasks to return their outputs and delegate serialization to a dedicated function.

For example, your task may look like this:

```
def my_function
    # no need to serialize here, simply return the output
    return  1  2  3
```

Important: Serialization only works on function tasks.

And your serializer may look like this:

```
@serializer         ='joblib'          = '.csv'  '.txt'
def my_serializer
    pass
```

Resources

1. A complete example.
2. An example showing tasks with a variable number of output files.
3. Serialization *User Guide* (explains the API step-by-step).

2.5.6 Database configuration

To have SQL scripts as tasks, you must configure a database client. There are two available clients: *ploomber.clients.SQLAlchemyClient* and *ploomber.clients.DBAPIClient*, we recommend using the sqlalchemy client if your database is supported because it is compatible with more types of SQL tasks (e.g., *ploomber.tasks.SQLDump*, which dumps data into a local file).

Using `SQLAlchemyClient`

Ensure that you can connect to the database using `sqlalchemy`:

```
from            import

      =               'DATABASE_URI'
```

`DATABASE_URI` depends on the type of database. sqlalchemy supports a wide range of databases; you can find a list in their documentation, while others come in third-party packages (e.g., Snowflake).

If `create_engine` is successful, ensure you can query your database:

```
with       .          as
           =     .        'SELECT * FROM some_table LIMIT 10'
```

If the query works, you can initialize a SQLAlchemyClient with the same `DATABASE_URI`:

```
from                 import

      =                  'DATABASE_URI'
```

Using Snowflake

Here's some sample code to configure Snowflake:

```
# install snowflake-sqlalchemy
pip install snowflake-sqlalchemy
```

Build your URL with the helper function:

```
from                      import

      =          ='user'
                     ='pass'
                   ='acct'
                     ='warehouse'
                   ='db'
                 ='schema'
                ='role'

      =                    **
```

If using OAuth instead of user/password authentication, you need to include the token:

```
import
import          # pip install requests
from                      import

def get_snowflake_token
            =  'content-type'  'application/x-www-form-urlencoded'
          =
        'grant_type'  'password'
        'scope'  'SESSION:ROLE-ANY'
        'username: username,
        'password'
        'client_id: f'      //         .                  .   ',

             =         .                    =             =
                                  =False

    return         .              .      'access_token'  .

      =                     'user'  'password'  'account'

       =          ='user'
                     ='acct'
                       ='warehouse'
                      ='db'
                    ='schema'
                   ='role'
                             ='oauth'
                    =

       =                       **
```

Using `DBAPIClient`

DBAPIClient takes a function that returns a DBAPI compatible connection and parameters to initialize such connection.

Here's an example with SQLite:

```
from                  import
import

       =                    .                     ='my.db'
```

Under the hood, Ploomber calls `sqlite3.connect(database='my.db')`.

Another example, this time using Snowflake:

```
from                  import
import

       =          ='USER'          ='PASS'         ='ACCOUNT'
       =                       .         .
```

Configuring the client in `pipeline.yaml`

Check out the *SQL Pipelines* to learn how to configure the database client in your `pipeline.yaml` file.

Examples

To see some examples using SQL connections, see this:

1. A short example that dumps data.
2. A SQL pipeline.
3. An ETL pipeline.

2.5.7 Task grids

You can use `tasks[*].grid` to create multiple tasks from a single task declaration, for example, to train various models with different parameters:

```
# execute independent tasks in parallel
executor  parallel

tasks
    source  random-forest.py
    # generates random-forest-5-gini, random-forest-10-gini, ..., random-forest-20-
↪entropy
    name  random-forest-[[n_estimators]]-[[criterion]]
    product  random-forest-[[n_estimators]]-[[criterion]].html
    grid
        # creates 6 tasks (3 * 2)
        n_estimators
        criterion
```

Download example:

```
pip install ploomber
ploomber examples -n cookbook/grid -o grid
   grid
pip install -r requirements.txt
ploomber build
```

Click here to see the complete example.

For full details, see the `grid` API *documentation*.

An in-depth tutorial showing how to use `grid` and MLflow for experiment tracking is available here.

2.5.8 Notebook Executors

Ploomber currently supports two notebook executors:

- Papermill
- Ploomber Engine

Papermill(default)

Papermill allows you to parameterizing, executing, and analyzing Jupyter notebooks. By default, if the `executor` argument is not specified in `NotebookRunner`, Ploomber will use Papermill to execute notebooks. Additionally, you can pass the following Papermill arguments in `executor_params` while using `NotebookRunner`.

Note: When migrating from one executor to another, make sure to check the parameters passed in `executor_params`, as different executors might have different arguments and functionalities.

Sample pipeline

```
tasks
    # By default, papermill engine is used for executing the scripts
    # source is the code you want to execute (.ipynb also supported)
    source  get.py
    # products are task's outputs
    product
      # scripts generate executed notebooks as outputs
      nb  output/1-get.html
      # you can define as many outputs as you want
      data  output/raw_data.csv
    # Selecting the executor for notebook
    executor  papermill
    # Executor params: Here passed to papermill
    executor_params
        log_output  True
```

Ploomber-Engine

Ploomber-Engine is a notebook executor developer by the Ploomber team with better support for debugging and deployment. You can use Ploomber-Engine with Ploomber by setting the `executor` argument to `ploomber-engine` in `NotebookRunner`. Additionally, you can pass the following arguments in `executor_params` while using `NotebookRunner` with `ploomber-engine`.

Sample pipeline

```
tasks
    # By default, papermill engine is used for executing the scripts
    # source is the code you want to execute (.ipynb also supported)
    source  get.py
    # products are task's outputs
    product
      # scripts generate executed notebooks as outputs
      nb  output/1-get.html
      # you can define as many outputs as you want
      data  output/raw_data.csv
```

(continues on next page)

(continued from previous page)

```
    # Selecting the executor for notebook
    executor  ploomber-engine
    # Executor params: Here passed to Ploomber-Engine
    executor_params
        log_output  True
```

2.5.9 Custom pipeline loading logic

Note: For a complete code example click here

When writing a pipeline via a `pipeline.yaml` file, the most common way to interact is via the command-line interface (e.g., calling `ploomber build`). However, we may want to add more logic or embed it as part of a script in some cases.

Ploomber represents pipelines using `DAG` objects. You can convert your `pipeline.yaml` to a `DAG` object in Python with the following code:

```
from               import

def load
         =         'pipeline.yaml'
        =     .
    return
```

The `spec` variable is an object of type *`ploomber.spec.DAGSpec`*, while `dag` is of type *`ploomber.DAG`*, to learn more about their interfaces, click on any of the links.

Assume the code above exists in a file named `pipeline.py`; you may load the pipeline from the CLI with:

```
ploomber build --entry-point pipeline.load
```

Or the shortcut:

```
ploomber build -e pipeline.load
```

The `load` function may contain extra logic; for example, you may skip tasks based on some custom rules or compute dynamic parameters. For examples with custom logic, click here.

2.5.10 Hooks (e.g., `on_finish`)

Hooks allow you to execute an arbitrary function when a task finishes:

1. `on_render` executes right before executing the task.
2. `on_finish` executes when a task finishes successfully.
3. `on_failure` executes when a task errors during execution.

Suppose your `pipeline.yaml` looks like this:

```
tasks
      source  tasks.my_task
      product  products/output.csv
```

(continues on next page)

(continued from previous page)

```
      on_render  hooks.on_render
      on_finish  hooks.on_finish
      on_failure  hooks.on_failure
```

And your `hooks.py` file looks like this:

```
def on_render
          'this runs before executing my_task'

def on_finish
          'this runs when my_task executes without errors!'

def on_failure
          'this runs when my_task raise an exception during execution!'
```

Hooks can take parameters; for example, you may add the `product` parameter to the hook, and Ploomber will call the hook with the `product` for the corresponding task. Adding arguments is useful when your hook needs information from the task. Furthermore, you can pass arbitrary parameters loaded from the `pipeline.yaml`. To learn more about hook parameters *click here*.

Tip: Developing hooks interactively

If you want to develop hooks interactively, you may start a session inside your hook like this:

```
def on_finish
    from         import
          'this runs when my_task executes without errors!'
```

Once you execute your pipeline, an interactive session will start. Interactive sessions are useful to explore the arguments (such as `product`) that your hook can request.

If you want to exclusively run the `on_finish` hook (and skip task's execution):

```
ploomber task {task-name} --on-finish
```

Resources

1. For a detailed look at the hooks API, *click here*.
2. For a tutorial, check out *Pipeline testing*, which is an in-depth guide of using `on_finish` for data testing.

DAG-level hooks

There are also DAG-level hooks, which work similarly. Declare them at the top section of your `pipeline.yaml` file:

```
# dag-level hooks
on_render  hooks.on_render
on_finish  hooks.on_finish
on_failure  hooks.on_failure

tasks
```

(continues on next page)

(continued from previous page)

```
    source  tasks.my_task
    product  products/output.csv
```

Click here to learn more.

2.6 API Reference

2.6.1 Spec API (`pipeline.yaml`)

Note: This document assumes you are already familiar with Ploomber's core concepts (DAG, product, task, and upstream). If you're not, check out this guide: *Basic concepts*.

This section describes the `pipeline.yaml` schema.

`meta`

`meta` is an optional section for meta-configuration, it controls how the DAG is constructed.

`meta.source_loader`

Load task sources (`tasks[*].source`) from a Python module. For example, say you have a module `my_module` and want to load sources from a `path/to/sources` directory inside that module:

```
meta
    source_loader
        module  my_module
        path  path/to/sources
```

`meta.import_tasks_from`

Add tasks defined in a different file to the current one. This direcive is useful for composing pipelines. For example, if you have a training and a serving pipeline, you can define the pre-processing logic in a `pipeline.preprocessing.yaml` and then import the file into `pipeline.training.yaml` and `pipeline.serving.yaml`:

```
meta
    import_tasks_from  /path/to/tasks.yaml
```

The file must be a list where each element is a valid Task.

Click here to see a batch serving example.

Click here to see an online serving example.

meta.extract_upstream

Extract upstream dependencies from the source code (`True` by default).

```
meta
    extract_upstream  True
```

If False, tasks must declare dependencies using the `upstream` key:

```
meta
    extract_upstream  false

tasks
      source  tasks/clean.py
      product  outupt/report.html
      upstream
```

meta.extract_product

Default:

```
meta
    extract_product  False
```

meta.product_default_class

Product class key for a given task class. Names should match (case-sensitive) the names in the Python API. These are rarely changed, except for `SQLScript`. Defaults:

```
meta
    product_default_class
        SQLScript  SQLRelation
        SQLDump  File
        NotebookRunner  File
        ShellScript  File
        PythonCallable  File
```

executor

Determines which executor to use:

1. `serial`: Runs one task at a time (Note: By default, function tasks run in a subprocess)
2. `parallel`: Run independent tasks in parallel (Note: this runs all tasks in a subprocess)
3. Dotted path: This allows you to customize the initialization parameters

For example, say you want to use the *ploomber.executors.Serial* executor but do not want to run functions in a subprocess, you can pass a dotted path and custom parameters like this:

```
executor
  dotted_path  ploomber.executors.Serial
  build_in_subprocess  false # do not run function tasks in a subprocess
```

Another common use case is to limit the number of subprocesses when using the *ploomber.executors.Parallel* executor:

```
executor
  dotted_path  ploomber.executors.Parallel
  processes  2 # limit to a max of 2 processes
```

You can set which method should be used to start child processes. method can be 'fork', 'spawn' or 'forkserver':

```
executor
  dotted_path  ploomber.executors.Parallel
  processes  2 # limit to a max of 2 processes
  start_method  spawn # start child process using 'spawn' method
```

To learn more about the executors:

- *ploomber.executors.Serial*
- *ploomber.executors.Parallel*

clients

These are the default clients. It allows you to specify a single client for all Tasks/Products for a given class. The most common use case is SQL database configuration.

Other scenarios are *ploomber.products.File* clients, which Ploomber can use to backup pipeline results (say, for example, you run a job that trains several models and want to save output results. You can use *ploomber.clients.GCloudStorageClient* or *ploomber.clients.S3Client* for that.

Keys must be valid `ploomber.tasks` or `ploomber.products` names, values must be dotted paths to functions that return a `ploomber.clients` instance.

Can be a string (call without arguments):

```
clients
    # this assumes there is a clients.py with a get_client function
                clients.get_client
```

Or a dictionary (to call with arguments):

```
clients

        # this assumes there is a clients.py with a get_client function
        dotted_path  clients.get_client
        kwarg_1  value_1
        ...
        kwarg_k  value_k
```

```
clients
  SQLDump  clients.get
```

(continues on next page)

(continued from previous page)

```
tasks
    source  query.sql
    product  output/data.csv
    # dump everything into a single file
    chunksize  null
```

```
from                  import


def get
    return                  'sqlite:///my.db'
```

Download:

```
ploomber examples -n cookbook/sql-dump -o sql-dump
```

```
clients
  # configures a dag-level File client
  File  clients.get_local # you can switch to clients.get_s3 or clients.get_gcloud

tasks
    source  functions.create_file
    # upon execution, this file is uploaded to storage
    product  products/some-file.txt

    source  scripts/some-script.py
    # upon execution, both files are uploaded to storage
    product
      nb  products/some-script.ipynb
      file  products/another-file.txt
    # you may also pass a task-level File client if you don't want to upload
    # all products in the pipeline
    # client: clients.get_s3
```

```
from                  import


def get_local
    """Returns local client
    """
    return                    'backup'


def get_s3
    """Returns S3 client
    """
    # assumes your environment is already configured, you may also pass the
    # json_credentials_path
    return                     ='some-bucket'        ='my-project/products'
```

(continues on next page)

(continued from previous page)

```
def get_gcloud
    """Returns google cloud storage client
    """
    # assumes your environment is already configured, you may also pass the
    # json_credentials_path
    return                                  ='some-bucket'
                                           ='my-project/products'
```

Download:

```
ploomber examples -n cookbook/file-client -o file-client
```

- SQL pipeline
- Example using BigQuery and Cloud Storage.

on_{render, finish, failure}

Important: Hooks are **not** executed when opening scripts/notebooks in *Jupyter*.

These are hooks that execute when specific events happen:

1. `on_render`: executes after verifying there are no errors in your pipeline declaration (e.g., a task that doesn't exist declared as an upstream dependency)
2. `on_finish`: executes upon successful pipeline run
3. `on_failure`: executes upon failed pipeline run

They all are optional and take a dotted path as an argument. For example, assume you have a `hooks.py` with function `on_render`, `on_finish`, and `on_failure`. You can add them to your `pipeline.yaml` like this:

```
on_render  hooks.on_render
on_finish  hooks.on_finish
on_failure  hooks.on_failure
```

If your hook takes arguments, you may call it like this:

```
# to call any hook with arguments
# {hook-name} must be one of: on_render, on_finish, on_failure

    dotted_path
    argument  value
```

Calling with arguments is useful when you have *a parametrized pipeline*.

If you need information from your DAG in your hook, you may request the `dag` (*ploomber.DAG*) argument in any of the hooks. `on_finish` can also request a `report` argument, which constains a summary report of the pipeline's execution.

`on_failure` can request a `traceback` argument which will have a dictionary, possible keys are `build` which has the build error traceback, and `on_finish` which includes the `on_finish` hook traceback, if any. For more information, see the DAG documentation *ploomber.DAG*.

```
# dag-level hooks
on_render
  dotted_path  hooks.dag_level_on_render
  my_param  10
on_finish  hooks.dag_level_on_finish
on_failure  hooks.dag_level_on_failure
```

```
def dag_level_on_render
    """Executed after the pipeline renders (before execution)
    """
          f'rendered DAG! my_param={        }'


def dag_level_on_finish
    """Executes after the pipeline runs all tasks
    """
          f'Finished executing pipeline {   }, report:\n{      }'


def dag_level_on_failure
    """Executes if the pipeline fails
    """
    if          .    'build'
              'Pipeline execution failed while running the tasks!'

    if          .    'on_finish'
              'Pipeline execution failed while executing an on_finish hook!'
```

Download:

```
ploomber examples -n cookbook/hooks -o hooks
```

`serializer` and `unserializer`

By default, tasks whose source is a function (i.e., *`ploomber.tasks.PythonCallable`*). Receive input paths (in `upstream`) and output paths (in `product`) when the function executes. Saving interim results allows Ploomber to provide incremental builds (*What are incremental builds?*).

However, in some cases, we might want to provide a pipeline that performs all operations in memory (e.g., to do online serving). *`ploomber.OnlineDAG`* can convert a file-based pipeline into an in-memory one without code changes, allowing you to re-use your feature engineering code for training and serving. The only requisite is for tasks to configure a `serializer` and `unserializer`. Click here to see an example.

Normally, a task whose source is a function looks like this:

```
import        as

def my_task
                =   .                  'name'
    # process data...
    # save product
              .
```

And you use the `product` parameter to save any task output.

However, if you add a `serializer`, `product` isn't passed, and you must return the product object:

```
import        as

def my_task
                =   .                  'name'
    # process data...
    return
```

The `serializer` function is called with the returned object as its first argument and `product` (output path) as the second argument:

```

```

A similar logic applies to `unserializer`; when present, the function is called for each upstream dependency with the product as the argument:

```

```

In your task function, you receive objects (instead of paths):

```
import        as

def my_task
    # no need to call pd.read_csv here
                =          'name'
    return
```

If you want to provide a Task-level serializer/unserializer pass it directly to the task, if you set a DAG-level serializer/unserializer and wish to exclude specific task pass `serializer: null` or `unserializer: null` in the selected task.

```
serializer  util.my_serializer
unserializer  util.my_unserializer

tasks
    source  tasks.one_product
    product  output/one.txt

    source  tasks.many_products
    product
      something  output/something.csv
      another  output/something.txt

    source  tasks.joblib_product
    product  output/something.joblib

    source  tasks.final_product
    product  output/final.csv
```

```
from            import
```

(continues on next page)

(continued from previous page)

```
@serializer         ='joblib'          = '.csv'  '.txt'
def my_serializer
    pass


@unserializer         ='joblib'          = '.csv'  '.txt'
def my_unserializer
    pass
```

Download:

```
ploomber examples -n cookbook/serialization -o serialization
```

source_loader

If you package your project (i.e., add a `setup.py`), `source_loader` offers a convenient way to load sources inside such package.

For example, if your package is named `my_package` and you want to load from the folder `my_sources/` within the package:

```
meta
    source_loader
        module  my_package
        path  my_sources

tasks
    # this is loaded from my_package (my_sources directory)
      source  script.sql
      # task definition continues...
```

To find out the location used, you can execute the following in a Python session:

```
import                                # print package location
```

The above should print something like `path/to/my_package/__init__.py`. Using the configuration above, it implies that source loader will load the file from `path/to/my_package/my_sources/script.sql`.

Note: this only applies to tasks whose `source` is a relative path. Dotted paths and absolute paths are not affected.

For details, see *ploomber.SourceLoader*, which is the underlying Python implementation. Here's an example that uses source_loader.

SQLScript product class

By default, SQL scripts use *ploomber.products.SQLRelation* as product class. Such product doesn't save product's metadata; required for incremental builds (*What are incremental builds?*). If you want to use them, you need to change the default value and configure the product's client.

Here's an example that uses `product_default_class` to configure a SQLite pipeline with incremental builds.

For more information on product clients, see: *FAQ and Glossary*.

Loading from a factory

The CLI looks for a `pipeline.yaml` by default, if you're using the Python API, and want to save some typing, you can specify a `pipeline.yaml` like this:

```
# pipeline.yaml
location
```

With such configuration, commands such as `ploomber build` will work.

task

`task` schema.

Tip: All other keys passed here are forwarded to the class constructor, so the allowed values will depend on the task class. For example, if running a notebook the task class is *ploomber.tasks.NotebookRunner*, if it's a function it'll be a *ploomber.tasks.PythonCallable*, see the documentation to learn what extra arguments they take.

---

tasks[*].name

The name of the task. The filename (without the extension) is used if not defined.

tasks[*].source

Indicates where the source code for a task is. This can be a path to a files if using scripts/notebooks or dotted paths if using a function.

By default, paths are relative to the `pipeline.yaml` parent folder (absolute paths are not affected), unless `source_loader` is configured; in such situation, paths are relative to the location configured in the `SourceLoader` object. See the `source_loader` section for more details.

For example, if your pipeline is located at `project/pipeline.yaml`, and you have:

```
tasks
      source  scripts/my_script.py
      # task definition continues...
```

Ploomber will expect your script to be located at `project/scripts/my_script.py`

If using a function, the dotted path should be importable. for example, if you have:

```
tasks
      source  my_package.my_module.my_function
      # task definition continues...
```

Ploomber runs a code equivalent to:

```
from                      import
```

tasks[*].product

Indicates output(s) generated by the task. This can be either a File(s) or SQL relation(s) (table or view). The exact type depends on the `source` value for the given task: SQL scripts generate SQL relations, everything else generates files.

When generating files, paths are relative to the `pipeline.yaml` parent directory. For example, if your pipeline is located at `project/pipeline.yaml`, and you have:

```
tasks
      source  scripts/my_script.py
      product  output/my_output.csv
```

Ploomber will save your output to `project/output/my_output.csv`

When generating SQL relations, the format is different:

```
tasks
      source  scripts/my_script.sql
      # list with three elements (last one can be table or view)
      product
      # schema is optional, it can also be: [name, table]
```

If the task generates multiple products, pass a dictionary:

```
tasks
      source  scripts/my_script.py
      product
        nb  output/report.html
        data  output/data.csv
```

Note: The name of keys in the product dictionary can be chosen freely so as to be descriptive of the outputs (e.g. `data`, `data_clean`, `model`, etc.)

The mechanism to make `product` available when executing your task depends on the type of task.

SQL tasks receive a `{{product}}` placeholder:

```
-- {{product}} is replaced by "schema.name" or "name" if schema is empty
CREATE TABLE             AS
SELECT * FROM          WHERE           > 10
```

If `product` is a dictionary, use `{{product['key']}}`

Python/R scripts/notebooks receive a `product` variable in the "injected-parameters" cell:

```
# %% tags=["parameters"]
        = None

# %% tags=["injected-parameters"]
        = '/path/to/output/data.csv'

# your code...
```

If `product` is a dictionary, this becomes `product = {'key': '/path/to/output/data.csv', ...}`

Python functions receive the `product` argument:

```
import        as

def my_task
    # process data...
      .
```

If `product` is a dictionary, use `product['key']`.

The same logic applies when making `upstream` dependencies available to tasks, but in this case. `upstream` is always a dictionary: SQL scripts can refer to their upstream dependencies using `{{upstream['key']}}`. While Python scripts and notebooks receive upstream in the "injected-parameters" cell, and Python functions are called with an `upstream` argument.

tasks[*].params

Use this section to pass arbitrary parameters to a task.

```
tasks
      source
      product
      params
        my_param  42
```

You can also generate parameters from functions, for example:

```
tasks
      source
      product
      params
        my_param  params::my_param_generate
```

This will `import params` and call the function `my_param_generate`. The returned value is assigned to `my_param`.

If your function takes parameters:

```
tasks
      source
      product
      params
        my_param
          dotted_path  params::my_param_generate
```

(continues on next page)

(continued from previous page)

```
          arg1  value1
          arg2  value2
```

In this case, `my_param_generate` is called like `my_param_generate(arg1='value1', arg2='value2')`

The mechanism to pass `params` to tasks depends on the task type:

SQL tasks receive them as placeholders.

```
-- {{my_param}} is replaced by 42
SELECT * FROM          WHERE           >
```

Python/R scripts/notebooks receive them in the "injected-parameters" cell:

```
# %% tags=["parameters"]
         = None

# %% tags=["injected-parameters"]
         = 42

# your code...
```

Python functions receive them as arguments:

```
# function is called with my_param=42
def my_task
    pass
```

Changed in version 0.21: Allows passing dotted paths (`module::function`) to `tasks[*].params`

tasks[*].on_{render, finish, failure}

Important: Hooks are **not** executed when opening scripts/notebooks in *Jupyter.*

These are hooks that execute under certain events. They are equivalent to *dag-level hooks*, except they apply to a specific task. There are three types of hooks:

1. `on_render` executes right before executing the task.
2. `on_finish` executes when a task finishes successfully.
3. `on_failure` executes when a task errors during execution.

They all are optional and take a dotted path as an argument. For example, assume your `hooks.py` with functions `on_render`, `on_finish`, and `on_failure`. You can add those hooks to a task in your `pipeline.yaml` like this:

```
tasks
      source  tasks.my_task
      product  products/output.csv
      on_render  hooks.on_render
      on_finish  hooks.on_finish
      on_failure  hooks.on_failure
```

If your hook takes arguments, you may call it like this:

```
# to call any hook with arguments
# {hook-name} must be one of: on_render, on_finish, on_failure

    dotted_path
    argument  value
```

Calling with arguments is useful when you have *a parametrized pipeline*.

If you need information from the task, you may add any of the following arguments to the hook:

1. `task`: Task object (a subclass of *`ploomber.tasks.Task`*)
2. `client`: Tasks's client (a subclass of *`ploomber.clients.Client`*)
3. `product`: Tasks's product (a subclass of *`ploomber.products.Product`*)
4. `params`: Tasks's params (a dictionary)

For example, if you want to check the data quality of a function that cleans some data, you may want to add an `on_finish` hook that loads the output and tests the data:

```
import        as

def on_finish
       =   .

    # check that column "age" has no NAs
    assert not   .   .      .
```

```
tasks
    source  tasks.do_something
    product  output/data.csv
    # task-level hooks
    on_render
      dotted_path  hooks.on_render
      my_param  20
    on_finish  hooks.on_finish
    on_failure  hooks.on_failure
```

```
def on_render
    """Executed after the task renders (before execution)
    """
          f'Finished rendering {    .    } with my_param {        }, '
          f'client {      }, product {       }, and task params {      }'


def on_finish
    """Executes after the task runs
    """
          f'Finished running {    .    } with client {      }, '
          f'product {       } and params {      }'


def on_failure
    """Executes if the task fails
```

(continues on next page)

(continued from previous page)

```
    """
          f'{    .    } with client {      }, '
          f'product {       } and params {      } failed!'
```

Download:

```
ploomber examples -n cookbook/hooks -o hooks
```

tasks[*].params.resources_

The `params` section contains an optional section called `resources_` (Note the trailing underscore). By default, Ploomber marks tasks as outdated when their parameters change; however, parameters in the `resources_` section work differently: they're marked as outdated when the contents of the file change. For example, suppose you're using a JSON file as a configuration source for a given task, and want to make Ploomber re-run a task if such file changes, you can do something like this:

```
tasks
      source  scripts/my-script.py
      product  report.html
      params
        resources_
            # whenever the JSON file changes, my-script.py runs again
            file  my-config-file.json
```

tasks[*].grid

Sometimes, you may want to run the same task over a set of parameters, `grid` allows you to do so. For example, say you want to train multiple models, each one with a different set of parameters:

```
tasks
    source  random-forest.py
    # name is required when using grid
    name  random-forest-
    product  random-forest.html
    grid
        # six tasks: 3 * 2
        n_estimators
        criterion
```

The spec above generates six tasks, one for each combination of parameters (`3 * 2`). In this example, products will be named `random-forest-X.html` where `X` goes from `0` to `5`. Similarly, task names will be *random-forest-X*. You can customize task names and product names to contain the corresponding parameter values, see the below sections for details.

Generating large grids dynamically

In cases where listing each parameter isn't feasible, you can write a function to produce all values (**Added in version 0.21**):

```
# grid.py (note: this can be any name you want)
```

(continues on next page)

(continued from previous page)

```
def generate_values
    return       10
```

Then call it in your `pipeline.yaml` like this:

```
tasks
    source  random-forest.py
    # name is required when using grid
    name  random-forest-
    product  random-forest.html
    grid
        n_estimators  grid::generate_values
        criterion
```

Note: This feature is available since version `0.19.8`; however, in version `0.21`, the format changed to `module::function`. From `0.19.8` to `0.20`, the format was `module.function`.

The above will generate 20 tasks (10 generate from `n_estimators` times 2 generated by `criterion`).

If the function takes parameters:

```
tasks
    source  random-forest.py
    # name is required when using grid
    name  random-forest-
    product  random-forest.html
    grid
      n_estimators
        dotted_path  grid::generate_values
        arg1  value1
        arg2  value2

      criterion
```

Templating output paths (products)

You can also customize the product outputs to organize them in different folders and names (**Added in version 0.17.2**):

```
tasks
    source  random-forest.py
    name  random-forest-
    product  'n_estimators=[[n_estimators]]/criterion=[[criterion]].html'
    grid
        n_estimators
        criterion
```

The example above will generate outputs by replacing the parameter values; for example, it will store the random forest with `n_estimators=5`, and `criterion=gini` at, `n_estimators=5/criterion=gini.html`. Note that this uses square brackets to differentiate them from regular placeholders when using an `env.yaml` file.

Templating name tasks

Similarly, you can also customize task names (**Added in version 0.19.8**):

```
tasks
    source  random-forest.py
    name  random-forest-[[n_estimators]]-[[criterion]]
    product  'n_estimators=[[n_estimators]]/criterion=[[criterion]].html'
    grid
        n_estimators
        criterion
```

The above will generate with task names `random-forest-5-gini`, `random-forest-10-gini`, etc.

Passing a list of grids

You may pass a list instead of a dictionary to use multiple sets of parameters:

```
tasks
    source  train-model.py
    name  train-model-
    product  train-model.html
    grid
        model_type
        n_estimators
        criterion

        model_type
        n_estimators
        learning_rate
```

Creating a task that depends on all grid-generated tasks

To create a task downstream to all tasks generated by `grid`, you can use a wildcard with the * character:

```
# make all tasks that start with train-model- dependencies of this task
        =  'train-model-*'
```

Examples and changelog

```
    source  scripts/fit.py
    # generates tasks fit-1, fit-2, etc
    name  fit-[[model_type]]-[[n_estimators]]-[[criterion]][[learning_rate]]
    # disabling static_analysis because the notebook does not have
    # a fixed set of parameters (depends on random-forest vs ada-boost)
    static_analysis  disable
    product
      nb  products/report-[[model_type]]-[[n_estimators]]-[[criterion]][[learning_rate]].
↪html
      model  products/model-[[model_type]]-[[n_estimators]]-[[criterion]][[learning_
↪rate]].pickle
    grid
      # generates 6 tasks (1 * 3 * 2)
        model_type
        n_estimators
        criterion

      # generates 6 tasks (1 * 3 * 2)
        model_type
```

(continues on next page)

(continued from previous page)

```
        n_estimators
        learning_rate
```

Download:

```
ploomber examples -n cookbook/grid -o grid
```

```
executor  parallel

tasks
    source  tasks/load.py
    product
      nb  products/load.html
      data  products/data.csv

    source  tasks/fit.py
    name  fit-[[model]]
    product
      nb  products/fit-[[model]].html
      model  products/model-[[model]].pkl
    grid
        model
        model_params
          # optimize over these parameters
            n_estimators
            criterion

        model
        model_params
          # optimize over these parameters
            kernel
            C
```

Download:

```
pip install ploomber
ploomber examples -n cookbook/nested-cv -o nested-cv
```

Changed in version 0.21: Dotted paths are required to have the `module::function` format, `module.function` is no longer allowed.

New in version 0.19.8: Pass dotted path (`module.function`) to generate large grids dynamically

New in version 0.19.8: Customize task names using placeholders `[[placeholder]]`

New in version 0.17.2: Use `params` and `grid` in the same task. Values in `params` are constant across the grid.

New in version 0.17.2: Customize the product paths with placeholders `[[placeholder]]`

tasks[*].client

Task client to use. By default, the class-level client in the `clients` section is used. This task-level value overrides it. Required for some tasks (e.g., `SQLScript`), optional for others (e.g., `File`).

Can be a string (call without arguments):

```
client  clients.get_db_client
```

Or a dictionary (to call with arguments):

```
client
    dotted_path  clients.get_db_client
    kwarg_1  value_1
    ...
    kwarg_k  value_k
```

tasks[*].product_client

Product client to use (to save product's metadata). Only required if you want to enable incremental builds (*What are incremental builds?*) if using SQL products. It can be a string or a dictionary (API is the same as `tasks[*].client`).

More information on product clients: *FAQ and Glossary*.

tasks[*].upstream

Dependencies for this task. Only required if `meta.extract_upstream=True`

```
tasks
    ...
    upstream
```

Example:

```
tasks
    source  scripts/my-script.py
    product  output/report.html
    upstream
```

tasks[*].class

Task class to use (any class from ploomber.tasks). You rarely have to set this, since it is inferred from `source`. For example, *ploomber.tasks.NotebookRunner* for `.py` and `.ipynb` files, *ploomber.tasks.SQLScript* for `.sql`, and *ploomber.tasks.PythonCallable* for dotted paths.

tasks[*].product_class

This takes any class name from `ploomber.products`. You rarely have to set this, since values from `meta.product_default_class` contain the typical values.

Parametrizing with env.yaml

In some situations, it's helpful to parametrize a pipeline. For example, you could run your pipeline with a sample of the data as a smoke test; to make sure it runs before triggering a run with the entire dataset, which could take several hours to finish.

To add parameters to your pipeline, create and `env.yaml` file next to your `pipeline.yaml`:

```
my_param  my_value
nested
    param  another_value
```

Then use placeholders in your `pipeline.yaml` file:

```
tasks
      source  module.function
      params
        my_param  '{{my_param}}'
        my_second_param  '{{nested.param}}'
```

In the previous example, `module.function` is called with `my_param='my_value'` and `my_second_param='another_value'`.

A common pattern is to use a pipeline parameter to change the location of `tasks[*].product`. For example:

```
tasks
      source  module.function
      # path determined by a parameter
      product  '{{some_directory}}/output.csv'

      source  my_script.sql
      # schema and prefix determined by a parameter
      product   '{{some_schema}}'  '{{some_prefix}}_name'
```

This can help you keep products generated by runs with different parameters in different locations.

These are the most common use cases, but you can use placeholders anywhere in your `pipeline.yaml` values (not keys):

```
tasks
      source  module.function
      # doesn't work
      '{{placeholder}}'  value
```

You can update your `env.yaml` file or switch them from the command-line to change the parameter values, run `ploomber build --help` to get a list of arguments you can pass to override the parameters defined in `env.yaml`.

Note that these parameters are constant (they must be changed explicitly by you either by updating the `env.yaml` file or via the command line), if you want to define dynamic parameters, you can do so with the Python API, check out this example for an example.

Re-using values in `env.yaml`

In version `0.20` and newer, you can refer to existing keys in your `env.yaml` and re-use them in upcoming values:

```
prefix  path/to/outputs
reports  '{{prefix}}/reports' # resolves to /path/to/outputs/reports
models  '{{prefix}}/models' # resolves to /path/to/outputs/models
```

Note that order matters; you can only refer to keys that have been defined earlier in the file.

Setting parameters from the CLI

Once you define pipeline parameters, you can switch them from the command line:

```
ploomber {command} --env--param value # note the double dash
```

For example:

```
ploomber build --env--param value
```

Default placeholders

There are a few default placeholders you can use in your `pipeline.yaml`, even if not defined in the `env.yaml` (or if you don't have a `env.yaml` altogether)

- `{{here}}`: Absolute path to the parent folder of `pipeline.yaml`
- `{{cwd}}`: Absolute path to the current working directory
- `{{root}}`: Absolute path to project's root folder. It is usually the same as `{{here}}`, except when the project is a package (i.e., it has `setup.py` file), in such a case, it points to the parent directory of the `setup.py` file.
- `{{user}}`: Current username
- `{{now}}`: Current timestamp in ISO 8601 format (*Added in Ploomber 0.13.4*)
- `{{git_hash}}`: git tag (if any) or git hash (*Added in Ploomber 0.17.1*)
- `{{git}}`: returns the branch name (if at the tip of it), git tag (if any), or git hash (*Added in Ploomber 0.17.1*)
- `{{env.ANY_ENV_VAR}}` environment variable present on the instance running the pipeline can be referenced using this syntax (*Added in Ploomber 0.21.7*)

When packaging a soopervisor docker image and no *env.yaml* file is defined a default one will be generated including some default placeholders such as *{{git}}*, *{{git_hash}}*. The reason being is that the package being copied to the docker image will not include *.git* and other ignored folders so at runtime *ploomber* won't have the information necessary to calculate the git hash, hence this is being pre-calculated during image build time from the parent pipeline repo.

A common use case for this is when passing paths to files to scripts/notebooks. For example, let's say your script has to read a file from a specific location. Using `{{here}}` turns path into absolute so you can ready it when using Jupyter, even if the script is in a different location than your `pipeline.yaml`.

By default, paths in `tasks[*].product` are interpreted relative to the parent folder of `pipeline.yaml`. You can use `{{cwd}}` or `{{root}}` to override this behavior:

```
tasks
      source  scripts/my-script.py
      product
        nb  products/report.html
        data  product/data.csv
      params
        # make this an absolute file so you can read it when opening
        # scripts/my-script.py in Jupyter
        input_path  '{{here}}/some/path/file.json'
```

For more on parametrized pipelines, check out the guide: *Parametrized pipelines*.

2.6.2 Command line interface

This document summarizes commonly used commands. To get full details, execute `ploomber --help` or `ploomber {command_name} --help`.

When applicable, we use this sample pipeline to demonstrate which tasks will be executed after issuing a given command:

Assume yellow tasks are outdated and green tasks are up-to-date.

Executed tasks are shown in blue and skipped tasks are shown in white in diagrams below.

Build pipeline

```
ploomber build
```

Execute your pipeline end-to-end and speed it up by skipping tasks whose source code has not changed.

(Skips `B2` because it's up-to-date)

Build pipeline (forced)

```
ploomber build --force
```

Execute all tasks regardless of status.

Build pipeline partially

```
ploomber build --partially C
```

Builds your pipeline until it reaches task named `C`.

(Skips `B2` because it's up-to-date)

(Skips `D` because it's not needed to build `C`)

To force the execution of tasks regardless of status, use the `--force/-f` option.

You can also select several tasks at the same time using wildcards:

```
ploomber build --partially 'fit-*' # note the single quotes
```

The previous command will execute all tasks with the `fit-*` prefix and all their upstream dependencies.

You may skip building upstream dependencies using the `--skip-upstream`

```
ploomber build --partially 'fit-*' --skip-upstream # note the single quotes
```

Note that the previous command fails if the upstream products of `fit-*` tasks do not exist yet.

Plot

```
ploomber plot
```

Creates a pipeline plot and stores it.

New in Ploomber 0.18.2: You can plot the pipeline without installing extra dependencies. `pygraphviz` is still supported but optional. To learn more, *see this*.

To include the task's products in the plot (only supported when using the `pygraphviz` backend):

```
ploomber plot --include-products
```

Status

```
ploomber status
```

Show a table with pipeline status. For each task: name, last execution time, status, product, docstring (first line) and file location.

Report

```
ploomber report
```

Create an HTML report and save it in a `pipeline.html` file. The file includes the pipeline plot and a table with a summary for each task.

Build a single task

```
ploomber task C
```

To force execution regardless of status use the `--force/-f` option.

Get task status

```
ploomber task task_name --status
```

If you also want to build the task, you must explicitly pass `--build`.

Task source code

```
ploomber task task_name --source
```

If you also want to build the task, you must explicitly pass `--build`.

Create new project

The `scaffold` command allows you to start a new project:

```
ploomber scaffold
```

The command above generates a project with sample pipeline. To create an empty project:

```
ploomber scaffold --empty
```

New in 0.16: You can pass a positional argument `ploomber scaffold myproject`.

Note that if you run this command in a folder that already has a `pipeline.yaml` in a *Default locations*, it will parse your pipeline declaration looking for declared tasks whose source code file does not exist and proceed to create them.

```
ploomber scaffold
```

If you'd like to package your project:

```
ploomber scaffold --package
```

After creating a project, you can install dependencies with the `ploomber install` command (to learn more: *install*).

For a tutorial on the `ploomber scaffold` command: *Scaffolding projects*.

`install`

`ploomber install` installs dependencies:

```
ploomber install
```

`ploomber install` installs dependencies using `pip` if a `requirements.txt` file exists or `conda`, if an `environment.yml` file exists.

New in 0.16: `ploomber install` has a few options to customize set up, run `ploomber install --help` to learn more.

New in 0.16: `ploomber install` will install dependencies in the current environment, you can request creating a virtual environment with the `--create-env` option, which will use venv or `conda` (if installed). Previously, it always created a new environment

Upon installation, `ploomber install` generates lock files that contain specific versions for all required packages. Lock files are useful for ensuring the stability of your project since upgrades to your dependencies may break your code. To install from lock files:

```
ploomber install --use-lock
```

New in 0.16: `ploomber install` (without arguments) will use lock files if they exist. Otherwise, it'll use regular files.

nb

`nb` is short for *notebook*. This command manages notebooks and scripts in your pipeline.

Inject cell to scripts and notebooks in your pipeline:

```
ploomber nb --inject
```

Enable opening `.py` as notebooks in JupyterLab with one click on the file:

```
ploomber nb --single-click
```

Re-format `.ipynb` notebooks as `.py` files with the percent format:

```
ploomber nb -f py:percent
```

Re-format `.py` files as `.ipynb` notebooks:

```
ploomber nb -f ipynb
```

The rest of the options are useful when using editors such as *VSCode or PyCharm* or when running old JupyterLab versions (`<2.x`). When using recent JupyterLab versions, script/notebooks management is automatically performed by the *Jupyter plug-in*.

Other commands are available, run `ploomber nb --help` to learn more.

Interactive sessions

To start an interactive session:

```
ploomber interact
```

Your pipeline is available in the `dag` variable. Refer to *`ploomber.DAG`* documentation for details.

Doing `dag['task_name']` returns a Task instance, all task instances have a common API, but there are a few differences. Refer to the tasks documentation for details: *Tasks*.

The CLI guide describes some of the most common use cases for interactive sessions: *Interactive sessions*.

Examples

To get a copy of the examples from the Github repository.

List examples:

```
ploomber examples
```

Get one:

```
ploomber examples --name {name}
```

To download in a specific location:

```
ploomber examples --name {name} --output path/to/dir
```

For a tutorial on the `ploomber examples` command: *Downloading templates*.

Default locations

If you don't pass the `--entry-point/-e` argument to the command line, Ploomber will try to find one automatically by searching for a `pipeline.yaml` file at the current directory and parent directories.

If no such file exists, it looks for a `setup.py`. If it exists, it searches for a `src/{pkg}/pipeline.yaml` file where `{pkg}` is a folder with any name. `setup.py` is only required for packaged projects.

If your pipeline has a different filename, you can create a `setup.cfg` file and indicate what file you want to set as default. Note that **changing the default affects both the command-line interface and the Jupyter plug-in**.

```
[ploomber]
entry-point = path/to/pipeline.yaml
```

Note that paths are relative to the parent directory of `setup.cfg`.

Alternatively, you may set the `ENTRY_POINT` environment variable to a different filename (e.g., `export ENTRY_POINT=pipeline.serve.yaml`). Note that this must be a filename, not a path to a file.

If you want to know which file will be used based on your project's layout:

```
ploomber status --help
```

Look at the `--entry-point` description in the printed output.

New in version 0.19.6: Support for switching entry point with a `setup.cfg` file

2.6.3 Python API

This section lists the available classes and functions in the Python API. If you're writing pipelines with the Spec API (e.g., `pipeline.yaml` file), you won't interact with this API directly. However, you may still want to learn about *ploomber.spec.DAGSpec* if you need to load your pipeline as a Python object.

For code examples using the Python API, click here.

DAG

DAG([name, clients, executor])	A collection of tasks with dependencies
OnlineModel(module)	A subclass of *ploomber.OnlineDAG* to provider a simpler interface for online DAGs whose terminal task calls `model.predict`.
OnlineDAG()	Execute partial DAGs in-memory.
DAGConfigurator([d])	An object to customize DAG behavior
InMemoryDAG(dag[, return_postprocessor])	Converts a DAG to a DAG-like object that performs all operations in memory (products are not serialized).

ploomber.DAG

class ploomber.**DAG**(*name=None*, *clients=None*, *executor='serial'*)

A collection of tasks with dependencies

Parameters

- **name** (*str, optional*) – A name to identify this DAG
- **clients** (*dict, optional*) – A dictionary with classes as keys and clients as values, can be later modified using dag.clients[dag] = client
- **differ** (*CodeDiffer*) – An object to determine whether two pieces of code are the same and to output a diff, defaults to CodeDiffer() (default parameters)
- **executor** (*str or ploomber.executors instance, optional*) – The executor to use (ploomber.executors.Serial and ploomber.executors.Parallel), is a string is passed ('serial' or 'parallel') the corresponding executor is initialized with default parameters

name

A name to identify the DAG

Type
str

clients

A class to client mapping

Type
dict

executor

Executor object to run tasks

Type
ploomber.Executor

on_render

Function to execute upon rendering. Can request a "dag" parameter.

Type
callable

on_finish

Function to execute upon execution. Can request a "dag" parameter and/or "report", which contains the report object returned by the build function.

Type
callable

on_failure

Function to execute upon failure. Can request a "dag" parameter and/or "traceback" which will contain a dictionary, possible keys are "build" which contains the build error traceback and "on_finish" which contains the on_finish hook traceback, if any.

Type
callable

serializer

Function to serialize products from PythonCallable tasks. Used if the task has no serializer. See `ploombe.tasks.PythonCallable` documentation for details.

Type
callable

unserializer

Function to unserialize products from PythonCallable tasks. Used if the task has no serializer. See `ploombe.tasks.PythonCallable` documentation for details.

Type
callable

Examples

Spec API:

```
pip install ploomber
ploomber examples -n guides/first-pipeline -o example
   example
pip install -r requirements.txt
ploomber build
```

Python API:

```
>>> from         import
>>> from          import
>>> from               import
>>> from                  import
>>> from                   import
>>>      =  "echo hi > {{product['first']}}; "
...         "echo bye > {{product['second']}}"
>>>   =      'script.sh' .
>>>     =             =                          =False
>>>         =  'first'       'first.txt'   'second'       'second.txt'
>>>       =                  'script.sh'               =         ='script'
>>> def my_task
...           =               'script'  'first'  .
...            =               'script'  'second'  .
...                  .                  + ' ' +
>>>          =                            'final.txt'      =
>>>       >>
PythonCallable: my_task -> File('final.txt')
>>>   =    .
```

Methods

build([force, show_progress, debug, ...])	Runs the DAG in order so that all upstream dependencies are run for every task
build_partially(target[, force, ...])	Partially build a dag until certain task
check_tasks_have_allowed_status(allowed, ...)	
close_clients()	Close all clients (dag-level, task-level and product-level)
get(k[,d])	
get_downstream(task_name)	Get downstream tasks for a given task name
items()	
keys()	
plot([output, include_products, backend, ...])	Plot the DAG
pop(name)	Remove a task from the dag
render([force, show_progress, remote])	Render resolves all placeholders in tasks and determines whether a task should run or not based on the task.product metadata, this allows up-to-date tasks to be skipped
status(**kwargs)	Returns a table with tasks status
to_markup([path, fmt, sections, backend])	Returns a str (md or html) with the pipeline's description
values()	

build(*force=False*, *show_progress=True*, *debug=None*, *close_clients=True*)

Runs the DAG in order so that all upstream dependencies are run for every task

Parameters

- **force** (*bool, default=False*) – If True, it will run all tasks regardless of status, defaults to False
- **show_progress** (*bool, default=True*) – Show progress bar
- **debug** (*'now' or 'later', default=None*) – If 'now', Drop a debugging session if building raises an exception. Note that this modifies the executor and temporarily sets it to Serial with subprocess off and catching exceptions/warnings off. Restores the original executor at the end. If 'later' it keeps the executor the same and serializes the traceback errors for later debugging

close_clients

[bool, default=True] Close all clients (dag-level, task-level and product-level) upon successful build

Notes

All dag-level clients are closed after calling this function

Changed in version 0.20: `debug` changed from True/False to 'now'/'later'/None

New in version 0.20: `debug` now supports debugging NotebookRunner tasks

Returns
A dict-like object with tasks as keys and dicts with task status as values

Return type
BuildReport

build_partially(*target*, *force=False*, *show_progress=True*, *debug=None*, *skip_upstream=False*)

Partially build a dag until certain task

Parameters

- **target** (*str*) – Name of the target task (last one to build). Can pass a wildcard such as 'tasks-*'
- **force** (*bool, default=False*) – If True, it will run all tasks regardless of status, defaults to False
- **show_progress** (*bool, default=True*) – Show progress bar
- **debug** (*'now' or 'later', default=None*) – If 'now', Drop a debugging session if building raises an exception. Note that this modifies the executor and temporarily sets it to Serial with subprocess off and catching exceptions/warnings off. Restores the original executor at the end. If 'later' it keeps the executor the same and serializes the traceback errors for later debugging
- **skip_upstream** (*bool, default=False*) – If False, includes all upstream dependencies required to build target, otherwise it skips them. Note that if this is True and it's not possible to build a given task (e.g., missing upstream products), this will fail

Notes

Changed in version 0.20: `debug` changed from True/False to 'now'/'later'/None

New in version 0.20: `debug` now supports debugging NotebookRunner tasks

check_tasks_have_allowed_status(*allowed*, *new_status*)

close_clients()

Close all clients (dag-level, task-level and product-level)

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_downstream(*task_name*)

Get downstream tasks for a given task name

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

plot(*output='embed'*, *include_products=False*, *backend=None*, *image_only=False*)

Plot the DAG

Parameters

- **output** (*str, default='embed'*) – Where to save the output (e.g., pipeline.png). If 'embed', it returns an IPython image instead.
- **include_products** (*bool, default=False*) – If False, each node only contains the task name, if True if contains the task name and products. Only available when using the pygraphviz backend
- **backend** (*str, default=None*) – How to generate the plot, if None it uses pygraphviz if installed, otherwise it uses D3 (which doesn't require extra dependencies), you can force to use a backend by passing 'pygraphviz', 'd3', or 'mermaid'.

pop(*name*)

Remove a task from the dag

render(*force=False*, *show_progress=True*, *remote=False*)

Render resolves all placeholders in tasks and determines whether a task should run or not based on the task.product metadata, this allows up-to-date tasks to be skipped

Parameters

- **force** (*bool, default=False*) – Ignore product metadata status and prepare all tasks to be executed. This option renders much faster in DAGs with products whose metadata is stored in remote systems, because there is no need to fetch metadata over the network. If the DAG won't be built, this option is recommended.
- **show_progress** (*bool, default=True*) – Show progress bar
- **remote** (*bool, default=False*) – Use remote metadata for determining task status. In most scenarios, you want this to be False, Ploomber uses this internally when exporting pipelines to other platforms (via Soopervisor).

status(***kwargs*)

Returns a table with tasks status

to_markup(*path=None*, *fmt='html'*, *sections=None*, *backend=None*)

Returns a str (md or html) with the pipeline's description

Parameters

sections (*list*) – Which sections to include, possible values are "plot", "status" and "source". Defaults to ["plot", "status"]

values() → an object providing a view on D's values

Attributes

clients
executor
product

ploomber.OnlineModel

class ploomber.**OnlineModel**(*module*)

A subclass of *ploomber.OnlineDAG* to provider a simpler interface for online DAGs whose terminal task calls `model.predict`. `OnlineModel` is initialized with a module following a standard structure. Looks for a `pipeline-features.yaml` in the module's root directory (e.g. `src/my_module/pipeline-features.yaml`), a `model.pickle` in the module's root directory. The terminal task is executed with a `model` parameter which contains the load model and calls `model.predict`. The last task in `pipeline-features.yaml` should be named `features`.

See here for a complete example: https://github.com/ploomber/projects/blob/master/templates/ml-online/src/ml_online/infer.py

Parameters

module – A module following a standard structure

Examples

```
>>> import
>>>       =
>>>      .          =
```

Methods

get_partial()	Must return the location of a partial dag (str or pathlib.Path)
init_dag_from_partial(partial)	Initialize partial returned by get_partial()
predict(**kwargs)	Returns the output of `model.predict(upstream['features'])`
terminal_params()	Must return a dictionary with parameters passed to `terminal_task`
terminal_task(upstream, model)	Las function to execute.

get_partial()

Must return the location of a partial dag (str or pathlib.Path)

classmethod init_dag_from_partial(*partial*)

Initialize partial returned by get_partial()

predict(***kwargs*)

Returns the output of `model.predict(upstream['features'])`

terminal_params()

Must return a dictionary with parameters passed to `terminal_task`

static terminal_task(*upstream*, *model*)

Las function to execute. The `upstream` parameter contains the output of all tasks that have no downstream dependencies

ploomber.OnlineDAG

class ploomber.**OnlineDAG**

Execute partial DAGs in-memory. This is an abstract class, to use it. Create a subclass and provide the required static methods.

See here for a complete example: https://github.com/ploomber/projects/blob/master/templates/ml-online/src/ml_online/infer.py

Methods

get_partial()	Must return the location of a partial dag (str or pathlib.Path)
init_dag_from_partial(partial)	Initialize partial returned by get_partial()
predict(**kwargs)	Run the DAG
terminal_params()	Must return a dictionary with parameters passed to `terminal_task`
terminal_task(upstream, model)	Las function to execute.

abstract static get_partial()

Must return the location of a partial dag (str or pathlib.Path)

classmethod init_dag_from_partial(*partial*)

Initialize partial returned by get_partial()

predict(***kwargs*)

Run the DAG

Parameters
****kwargs** – One parameter per root task (task with no upstream dependencies) in the partial DAG.

Returns
A dictionary with {task_name

Return type
returned_value}

abstract static terminal_params()

Must return a dictionary with parameters passed to `terminal_task`

abstract static terminal_task(*upstream*, *model*)

Las function to execute. The `upstream` parameter contains the output of all tasks that have no downstream dependencies

ploomber.DAGConfigurator

class ploomber.**DAGConfigurator**(*d=None*)

An object to customize DAG behavior

Note: this API is experimental an subject to change

To keep the DAG API clean, only the most important parameters are included in the constructor, the rest are accesible via a DAGConfigurator object

Available parameters:

outdated_by_code: whether source code differences make a task outdated cache_rendered_status: keep results from dag.render() whenever are needed again (e.g. when calling dag.build()) or compute it again every time.

cache_rendered_status: If True, once the DAG is rendered, subsequent calls to render will not do anything (rendering is implicitely called in build, plot, status), otherwise it will always render again.

hot_reload: Reload sources whenever they are updated

Examples

```
>>> from           import
>>>               =
>>>              .       .                  = True
>>>              .       .                      = False
>>>              .       .           = True
>>>     =              .
```

Methods

create(*args, **kwargs)	Return a DAG with the given parameters

create(**args*, ***kwargs*)

Return a DAG with the given parameters

***args, ****kwargs**
Parameters to pass to the DAG constructor

Attributes

params

ploomber.InMemoryDAG

class ploomber.**InMemoryDAG**(*dag*, *return_postprocessor=None*)

Converts a DAG to a DAG-like object that performs all operations in memory (products are not serialized). For this to work all tasks must be `PythonCallable` objects initialized with callables that return a value with valid `serializer` and `unserializer` parameters.

Parameters

dag (`ploomber.DAG`) – The DAG to use

Examples

```
"""
This example shows how to re-use the same feature engineering code in
both training (batch processing) and serving (online).
"""

import
from         import

import        as
from         import
from              import

from          import
from                import
from                   import
from                    import
from                import


def get
    """Get training data"""
      =         .                 =True
       =   "data"
       "target"  =   "target"
    return


# NOTE: "upstream" is the output from the task that executes before this one
def a_feature
    """Compute one feature"""
       =          "get"
    return   .           "a_feature"     "sepal length (cm)"  ** 2


def another
    """Compute another feature"""
       =          "get"
    return   .           "another"     "sepal width (cm)"  ** 2
```

(continues on next page)

(continued from previous page)

```
def join
    return          "get" .             "a_feature"  .             "another"


# NOTE: "product" is the model file output location
def fit
    """Train a model and save it (pickle format)"""
        =
       =   .                      "join"
      =   .     "target"      ="columns"
      =    "target"
       .

    with                    "wb"  as
              .


# NOTE: serializer and unserializer are special function that tell the pipeline
# how to convert the object returned by our tasks (pandas.DataFrame) to files.
# These are only required when we want to build a dag that works both in
# batch-processing and online mode
def serializer
    """Save all data frames as CSVs"""
        =

    # make sure the parent folder exists
             .      .            =True          =True

      .                =False


def unserializer
    """Function to read CSVs"""
    return   .


def add_features
    """
    Given a DAG, adds feature engineering tasks. The DAG must have a task "get"
    that returns the input data.
    """
             =     "get"

           =      "output"

    # instantiate tasks
                  =

                   / "a_feature.csv"

                 =
                   =
```

(continues on next page)

(continued from previous page)

```
                =

                    / "another.csv"

                  =
                    =

              =

                    / "join.csv"

                  =
                    =


    # establish dependencies
             >>
             >>
              +               +              >>

    return


def make_training
    """Instantiates the training DAG"""
    # setting build_in_subprocess=False because Python does not like when we
    # use multiprocessing in functions defined in the main module. Works if
    # we define them in a different one
        =             =                          =False

           =      "output"

    # add "get" task that returns the training data


                    / "get.csv"

                  =
                    =


    # add features tasks


    # add "fit" task for model training
          =                                  / "model.pickle"

    # train after joining features
        "join"  >>

    return
```

(continues on next page)

(continued from previous page)

```
def predict
    """Make a prediction after computing features"""
    return      .                 "join"


def validate_input_data
    """
    Validate input data
    """
         =
        "sepal length (cm)"
        "sepal width (cm)"
        "petal length (cm)"
        "petal width (cm)"


    if        .         !=
        raise ValueError f"Unexpected set of columns, expected: {    !r}"

                =    .      < 0
               =                                .

    if
        raise ValueError
            f"Column(s) {          !r} have one or more invalid"
            " (negative) observations"


    return


def make_predict
    """Instantiate a prediction DAG using a previously trained model"""
             =

    # this special function adds a task with name "get" that will just forward
    # whatever value we pass when calling .build(). You can pass a function
    # in the "preprocessor" argument to perform arbitrary logic like parsing
    # or validation
                         =             ="get"             =

    # we re-use the same code that we used for training!


    # load model generated by the training graph
    with           "output"  "model.pickle"   "rb"  as
              =       .

    # add the final task, this special function just executes whatever
    # function we pass as the first argument, we can pass arbitrary parameters
```

(continues on next page)

(continued from previous page)

```
    # using "params"
                 =
                    =              ="predict"        =          =


    # predict after joining features
             "join"  >>

    # convert our batch-processing pipeline to a in-memory one and return
    return


# instantiate training pipeline
    =
# run it (generates model.pkl)
   .

# instantiate prediction pipeline
         =

# input data: generates features from this and then feeds the model
             =   .

        "sepal length (cm)"   5.9
        "sepal width (cm)"   3
        "petal length (cm)"   5.1
        "petal width (cm)"   1.8



# pass input data through the prediction pipeline. A pipeline might have
# multiple inputs, in our case it only has one. The format is:
# {task_name: input_data}
       =         .       "get"

# result is a dictionary with {task_name: output}. Get the output from the
# predict task
       "predict"
```

Methods

build(input_data[, copy])	Run the DAG

build(*input_data*, *copy=False*)

Run the DAG

Parameters

- **input_data** (*dict*) – A dictionary mapping root tasks (names) to dict params. Root tasks are tasks in the DAG that do not have upstream dependencies, the corresponding dictionary is passed to the respective task source function as keyword arguments

- **copy** (*bool or callable*) – Whether to copy the output of an upstream task before passing it to the task being processed. It is recommended to turn this off for memory efficiency but if the tasks are not pure functions (i.e. mutate their inputs) this migh lead to bugs, in such case, the best way to fix it would be to make all your tasks pure functions but you can enable this option if memory consumption is not a problem. If True it uses the `copy.copy` function before passing the upstream products, if you pass a callable instead, such function is used (for example, you may pass `copy.deepcopy`)

Returns
A dictionary mapping task names to their respective outputs

Return type
dict

Tasks

Task(product, dag[, name, params])	Abstract class for all Tasks
PythonCallable(source, product, dag[, name, ...])	Execute a Python function
NotebookRunner(source, product, dag[, name, ...])	Run a Jupyter notebook using papermill.
ScriptRunner(source, product, dag[, name, ...])	Similar to NotebookRunner, except it uses python to run the code, instead of papermill, hence, it doesn't generate an output notebook.
SQLScript(source, product, dag[, name, ...])	Execute a script in a SQL database to create a relation or view
SQLDump(source, product, dag[, name, ...])	Dumps data from a SQL SELECT statement to a file(s)
SQLTransfer(source, product, dag[, name, ...])	Transfers data from a SQL database to another (Note: this relies on pandas, only use it for small to medium size datasets)
SQLUpload(source, product, dag[, name, ...])	Upload data to a SQL database from a parquet or a csv file.
PostgresCopyFrom(source, product, dag[, ...])	Efficiently copy data to a postgres database using COPY FROM (faster alternative to SQLUpload for postgres).
ShellScript(source, product, dag[, name, ...])	Execute a shell script.
DownloadFromURL(source, product, dag[, ...])	Download a file from a URL (uses urllib.request.urlretrieve)
Link(product, dag, name)	A dummy Task used to "plug" an external Product to a pipeline, this task is always considered up-to-date
Input(product, dag, name)	A dummy task used to represent input provided by the user, it is always considered outdated.

ploomber.tasks.Task

class ploomber.tasks.**Task**(*product*, *dag*, *name=None*, *params=None*)

Abstract class for all Tasks

Parameters

- **source** (*str or pathlib.Path*) – Source code for the task, for tasks that do not take source code as input (such as PostgresCopyFrom), this can be another thing. The source can be a template and can make references to any parameter in "params", "upstream" parameters or its own "product", not all Tasks have templated source (templating code is mostly used by Tasks that take SQL source code as input)

- **product** (Product) – The product that this task will create upon completion
- **dag** (DAG) – The DAG holding this task
- **name** (*str*) – A name for this task, if None a default will be assigned
- **params** (*dict*) – Extra parameters passed to the task on rendering (if templated source) or during execution (if not templated source)

params

A read-only dictionary-like object with params passed, after running 'product' and 'upstream' are added, if any

Type
Params

on_render

Function to execute after rendering. The function can request any of the following arguments: task, client, product, and params.

Type
callable

on_finish

Function to execute upon execution. Can request the same arguments as the on_render hook.

Type
callable

on_failure

Function to execute upon failure. Can request the same arguments as the on_render hook.

Type
callable

Notes

All subclasses must implement the same constuctor to keep the API consistent, optional parameters after "params" are ok

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

abstract run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.PythonCallable

class ploomber.tasks.**PythonCallable**(*source*, *product*, *dag*, *name=None*, *params=None*, *unserializer=None*, *serializer=None*, *debug_mode=None*)

Execute a Python function

Parameters

- **source** (*callable*) – The callable to execute
- **product** (ploomber.products.Product) – Product generated upon successful execution
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag
- **params** (*dict*) – Parameters to pass to the callable, by default, the callable will be executed with a "product" (which will contain the product object). It will also include a "upstream" parameter if the task has upstream dependencies along with any parameters declared here

- **unserializer** (*callable, optional*) – A callable to unserialize upstream products, the product object is passed as unique argument. If None, the source function receives the product object directly. If the task has no upstream dependencies, this argument has no effect
- **serializer** (*callable, optional*) – A callable to serialize this task's product, must take two arguments, the first argument passed is the value returned by the task's source, the second argument is the product oject. If None, the task's source is responsible for serializing its own product. If used, the source function must not have a "product" parameter but return its result instead
- **debug_mode** (*None, 'now' or 'later', default=None*) – If 'now', runs notebook in debug mode, this will start debugger if an error is thrown. If 'later', it will serialize the traceback for later debugging. (Added in 0.20)

Examples

Spec API:

```
tasks
    source  my_functions.my_task
    product  data.csv
```

```
# content of my_functions.py
from         import

def my_task
                 .
```

Spec API (multiple outputs):

```
tasks
    source  my_functions.another_task
    product
        one  one.csv
        another  another.csv
```

```
# content of my_functions.py
from         import

def another_task
                'one'  .
                'another'  .
```

Python API:

```
>>> from         import
>>> from          import
>>> from               import
>>> from                  import
>>> from                   import
>>>     =             =                          =False
>>> def my_function
...     # create data.csv
```

(continues on next page)

(continued from previous page)

```
...                 .
>>>                                 'data.csv'      =
PythonCallable: my_function -> File('data.csv')
>>>         =    .
```

Python API (multiple products):

```
>>> from         import
>>> from          import
>>> from               import
>>> from                  import
>>> from                   import
>>>     =             =                          =False
>>> def my_function
...                   'first'  .
...                   'second'  .
>>>         =  'first'       'first.csv'
...             'second'       'second.csv'
>>>      =                                        =
>>>         =    .
```

Notes

New in version 0.20: `debug` constructor flag renamed to `debug_mode` to avoid conflicts with the `debug` method.

More examples using the Python API.

The `executor=Serial(build_in_subprocess=False)` argument is only required if copy-pasting the example in a Python session. If you store the code in a script, you may delete it and call `dag.build` like this:

```
if          == '__main__'
       .
```

Then call your script:

```
python script.py
```

Methods

build([force, catch_exceptions])	Build a single task
debug([kind])	Run callable in debug mode.
load([key])	Loads the product.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug(*kind='ipdb'*)

Run callable in debug mode.

Parameters
kind (*str ('ipdb' or 'pdb')*) – Which debugger to use 'ipdb' for IPython debugger or 'pdb' for debugger from the standard library

Notes

Be careful when debugging tasks. If the task has run successfully, you overwrite products but don't save the updated source code, your DAG will enter an inconsistent state where the metadata won't match the overwritten product.

load(*key=None*, ***kwargs*)

Loads the product. It uses the unserializer function if any, otherwise it tries to load it based on the file extension

Parameters

- **key** – Key to load, if this task generates more than one product
- ****kwargs** – Arguments passed to the unserializer function

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
debug_mode	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.NotebookRunner

class ploomber.tasks.**NotebookRunner**(*source*, *product*, *dag*, *name=None*, *params=None*, *executor='papermill'*, *executor_params=None*, *papermill_params=None*, *kernelspec_name=None*, *nbconvert_exporter_name=None*, *ext_in=None*, *nb_product_key='nb'*, *static_analysis='regular'*, *nbconvert_export_kwargs=None*, *local_execution=False*, *check_if_kernel_installed=True*, *debug_mode=None*)

Run a Jupyter notebook using papermill. Support several input formats via jupytext and several output formats via nbconvert

Parameters

- **source** (*str or pathlib.Path*) – Notebook source, if str, the content is interpreted as the actual notebook, if pathlib.Path, the content of the file is loaded. When loading from a str, ext_in must be passed
- **product** (*ploomber.File*) – The output file
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str, optional*) – A str to indentify this task. Should not already exist in the dag
- **params** (*dict, optional*) – Notebook parameters. This are passed as the "parameters" argument to the papermill.execute_notebook function, by default, "product" and "upstream" are included
- **executor** (*str, optional*) – executor to use. Currently supports "ploomber-engine" and "papermill". Defaults to papermill executor. Can also be passed as "engine_name" in executor_params
- **executor_params** (*dict, optional*) – Parameters passed to executor, defaults to None. Please refer to each executor execute_notebook APIs to learn more about this.
- **papermill_params** (*dict, optional*) – Other parameters passed to papermill.execute_notebook, defaults to None
- **kernelspec_name** (*str, optional*) – Kernelspec name to use, if the file extension provides with enough information to choose a kernel or the notebook already includes kernelspec data (in metadata.kernelspec), this is ignored, otherwise, the kernel is looked up using jupyter_client.kernelspec.get_kernel_spec
- **nbconvert_exporter_name** (*str or dict, optional*) – Once the notebook is run, this parameter controls whether to export the notebook to a different parameter using the nbconvert package, it is not needed unless the extension cannot be used to infer the final output format, in which case the nbconvert.get_exporter is used. If nb_product_key is a list of multiple nb products keys, nbconvert_exporter_name should be a dict containing keys from this list.
- **ext_in** (*str, optional*) – Source extension. Required if loading from a str. If source is a pathlib.Path, the extension from the file is used.
- **nb_product_key** (*str or list, optional*) – If the notebook is expected to generate other products, pass the key to identify the output notebook (i.e. if product is a list with 3 ploomber.File, pass the index pointing to the notebook path). If the only output is the notebook itself, this parameter is not needed If multiple notebook conversions are required like html, pdf, this parameter should be a list of keys like 'nb_ipynb', 'nb_html, 'nb_pdf'.

- **static_analysis** (*('disabled', 'regular', 'strict'), default='regular'*) – Check for various errors in the notebook. In 'regular' mode, it aborts execution if the notebook has syntax issues, or similar problems that would cause the code to break if executed. In 'strict' mode, it performs the same checks but raises an issue before starting execution of any task, furthermore, it verifies that the parameters cell and the params passed to the notebook match, thus, making the notebook behave like a function with a signature.
- **nbconvert_export_kwargs** (*dict*) – Keyword arguments to pass to the `nbconvert.export` function (this is only used if exporting the output ipynb notebook to another format). You can use this, for example, to hide code cells using the exclude_input parameter. See `nbconvert` documentation for details. Ignored if the product is file with .ipynb extension.
- **local_execution** (*bool, optional*) – Change working directory to be the parent of the notebook's source. Defaults to False. This resembles the default behavior when running notebooks interactively via *jupyter notebook*
- **debug_mode** (*None, 'now' or 'later', default=None*) – If 'now', runs notebook in debug mode, this will start debugger if an error is thrown. If 'later', it will serialize the traceback for later debugging. (Added in 0.20)

Examples

Spec API:

```
tasks
    source  nb.ipynb
    product  report.html
```

Spec API (multiple outputs):

```
tasks
    source  nb.ipynb
    product
        # generated automatically by ploomber
        nb  report.html
        # must be generated by nb.ipynb
        data  data.csv
```

Spec API (multiple notebook products, added in `0.19.6`):

(generate the executed notebooks in multiple formats)

```
tasks
    source  script.py
    # keys can be named as per user's choice. None
    # of the keys are mandatory. However, every key mentioned
    # in this list should be a part of the product dict below.
    nb_product_key
    # When nb_product_key is a list, nbconvert_exporter_name
    # should be a dict with required keys from nb_product_key
    # only. If missing, it uses the default exporter
    nbconvert_exporter_name
        nb_pdf  webpdf
    # Every notebook product defined here should correspond to key
```

(continues on next page)

(continued from previous page)

```
    # defined in nb_product_key.
    product
        nb_ipynb  nb.ipynb
        nb_pdf  doc.pdf
        nb_html  report.html
        # must be generated by nb.ipynb
        data  data.csv
```

Python API:

```
>>> from         import
>>> from          import
>>> from               import
>>> from                  import
>>>     =
>>>                     'nb.ipynb'        'report.html'      =
NotebookRunner: nb -> File('report.html')
>>>    .
```

Python API (customize output notebook):

```
>>> from         import
>>> from          import
>>> from               import
>>> from                  import
>>>     =
>>> # do not include input code (only cell's output)
>>>                     'nb.ipynb'        'out-1.html'      =
...                                      = 'exclude_input'  True
...                    ='one'
NotebookRunner: one -> File('out-1.html')
>>> # Selectively remove cells with the tag "remove"
>>>        =  'TagRemovePreprocessor'   'remove_cell_tags'   'remove'
...        'HTMLExporter'
...          'preprocessors'
...     'nbconvert.preprocessors.TagRemovePreprocessor'
>>>                     'nb.ipynb'        'out-2.html'      =
...                                      = 'config'
...                    ='another'
NotebookRunner: another -> File('out-2.html')
>>>    .
```

Notes

Changed in version 0.22.4: Added native ploomber-engine support with *executor* parameter

Changed in version 0.20: `debug` constructor flag renamed to `debug_mode` to prevent conflicts with the `debug` method

Changed in version 0.19.6: Support for generating output notebooks in multiple formats, see example above.

nbconvert's documentation

Methods

build([force, catch_exceptions])	Build a single task
debug([kind])	Opens the notebook (with injected parameters) in debug mode in a temporary location
load([key])	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug(*kind='ipdb'*)

Opens the notebook (with injected parameters) in debug mode in a temporary location

Parameters
kind (*str, default='ipdb'*) – Debugger to use, 'ipdb' to use line-by-line IPython debugger, 'pdb' to use line-by-line Python debugger or 'pm' to to post-portem debugging using IPython

Notes

Be careful when debugging tasks. If the task has run successfully, you overwrite products but don't save the updated source code, your DAG will enter an inconsistent state where the metadata won't match the overwritten product.

load(*key=None*, ***kwargs*)

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
debug_mode	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
static_analysis	
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.ScriptRunner

class ploomber.tasks.**ScriptRunner**(*source*, *product*, *dag*, *name=None*, *params=None*, *ext_in=None*, *static_analysis='regular'*, *local_execution=False*)

Similar to NotebookRunner, except it uses python to run the code, instead of papermill, hence, it doesn't generate an output notebook. But it also works by injecting a cell into the source code. Source can be a `.py` script or an `.ipynb` notebook. **Does not support magics.**

Parameters

- **source** (*str or pathlib.Path*) – Script source, if str, the content is interpreted as the actual script, if pathlib.Path, the content of the file is loaded. When loading from a str, ext_in must be passed
- **product** (*ploomber.File*) – The output file
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str, optional*) – A str to indentify this task. Should not already exist in the dag
- **params** (*dict, optional*) – Script parameters. This are passed as the "parameters" argument to the papermill.execute_notebook function, by default, "product" and "upstream" are included

- **ext_in** (*str, optional*) – Source extension. Required if loading from a str. If source is a `pathlib.Path`, the extension from the file is used.
- **static_analysis** (*('disabled', 'regular', 'strict'), default='regular'*) – Check for various errors in the script. In 'regular' mode, it aborts execution if the notebook has syntax issues, or similar problems that would cause the code to break if executed. In 'strict' mode, it performs the same checks but raises an issue before starting execution of any task, furthermore, it verifies that the parameters cell and the params passed to the notebook match, thus, making the script behave like a function with a signature.
- **local_execution** (*bool, optional*) – Change working directory to be the parent of the script source. Defaults to False.

Examples

Spec API:

```
tasks
    source  script.py
    class  ScriptRunner
    product
        data  data.csv
        another  another.csv
```

Python API:

```
>>> from         import
>>> from          import
>>> from               import
>>> from                  import
>>>     =
>>>         =  'data'       'data.csv'   'another'       'another.csv'
>>>   =                  'script.py'              =
>>>   =    .
```

Methods

build([force, catch_exceptions])	Build a single task
debug([kind])	Opens the notebook (with injected parameters) in debug mode in a temporary location
load([key])	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug(*kind='ipdb'*)

Opens the notebook (with injected parameters) in debug mode in a temporary location

Parameters
kind (*str, default='ipdb'*) – Debugger to use, 'ipdb' to use line-by-line IPython debugger, 'pdb' to use line-by-line Python debugger or 'pm' to to post-portem debugging using IPython

Notes

Be careful when debugging tasks. If the task has run successfully, you overwrite products but don't save the updated source code, your DAG will enter an inconsistent state where the metadata won't match the overwritten product.

load(*key=None*, ***kwargs*)

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
static_analysis	
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.SQLScript

class ploomber.tasks.**SQLScript**(*source*, *product*, *dag*, *name=None*, *client=None*, *params=None*)

Execute a script in a SQL database to create a relation or view

Parameters

- **source** (*str or pathlib.Path*) – SQL script source, if str, the content is interpreted as the actual script, if pathlib.Path, the content of the file is loaded
- **product** (*ploomber.products.product*) – Product generated upon successful execution
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag
- **client** (*ploomber.clients.{SQLAlchemyClient, DBAPIClient}, optional*) – The client used to connect to the database. Only required if no dag-level client has been declared using dag.clients[class]
- **params** (*dict, optional*) – Parameters to pass to the script, by default, the callable will be executed with a "product" (which will contain the product object). It will also include a "upstream" parameter if the task has upstream dependencies along with any parameters declared here. The source code is converted to a jinja2.Template for passing parameters, refer to jinja2 documentation for details

Examples

Spec API:

```
clients
  SQLScript  clients.get
  SQLiteRelation  clients.get

tasks
    source  script.sql
    product
```

Python API (SQLite):

```
>>> import
>>> import        as
>>> from          import
>>> from                  import
>>> from                import
>>> from                  import
>>>         =        .               ='my_db.db'
>>>    =   .           'a'        100   'b'        100
>>>   =    .       'numbers'               =False
>>>     =
>>>        =                    .                    ='my_db.db'
...                                  =';'
>>>    .                   =
>>>    .                        =
>>>        =  'DROP TABLE IF EXISTS {{product}};'
...           'CREATE TABLE {{product}} AS '
```

(continues on next page)

(continued from previous page)

```
...             'SELECT * FROM numbers LIMIT 3'
>>>   =                                    'subset'  'table'
...                   =         ='create-subset'
>>>   =    .
>>>    =   .         'SELECT * FROM subset'
>>>        .
>>>   .     3
   a  b
0  0  0
1  1  1
2  2  2
```

See also:

ploomber.clients.SQLDump
A task to execute a SELECT statement and dump the output into a file

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load([limit])	Load this task's product in a pandas.DataFrame
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load(*limit=10*)

Load this task's product in a pandas.DataFrame

Parameters

limit (*int, default=10*) – How many records to load, defaults to 10

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.SQLDump

class ploomber.tasks.**SQLDump**(*source*, *product*, *dag*, *name=None*, *client=None*, *params=None*, *chunksize=10000*, *io_handler=None*)

Dumps data from a SQL SELECT statement to a file(s)

Parameters

- **source** (*str or pathlib.Path*) – SQL script source, if str, the content is interpreted as the actual script, if pathlib.Path, the content of the file is loaded
- **product** (*ploomber.products.product*) – Product generated upon successful execution
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag
- **client** (*ploomber.clients.{SQLAlchemyClient, DBAPIClient}, optional*) – The client used to connect to the database. Only required if no dag-level client has been declared using dag.clients[class]
- **params** (*dict, optional*) – Parameters to pass to the script, by default, the callable will be executed with a "product" (which will contain the product object). It will also include a "upstream" parameter if the task has upstream dependencies along with any parameters declared here. The source code is converted to a jinja2.Template for passing parameters, refer to jinja2 documentation for details

- **chunksize** (*int, optional*) – Number of rows per file, otherwise download the entire dataset in a single one. If not None, the product becomes a directory
- **io_handler** (*ploomber.io.CSVIO or ploomber.io.ParquetIO, optional*) – io handler to use (which controls the output format), currently only csv and parquet are supported. If None, it tries to infer the handler from the product's extension if that doesn't work, it uses io.CSVIO

Examples

Spec API:

```
clients
  # define a get function in clients.py that returns the client
  SQLDump  clients.get

tasks
  # script with a SELECT statement
    source  script.sql
    product  data.parquet
```

Full spec API example.

Python API:

```
>>> import
>>> import        as
>>> from          import
>>> from                  import
>>> from               import
>>> from                 import
>>>         =        .               ='my_db.db'
>>>    =    .          'a'        100   'b'        100
>>>   =   .       'numbers'               =False
>>>        .
>>>     =
>>>        =                    .                       ='my_db.db'
>>>   =         'SELECT * FROM numbers'       'data.parquet'
...                =         ='dump'        =                 =None
>>>   =    .
>>>    =   .             'data.parquet'
>>>   .     3
   a  b
0  0  0
1  1  1
2  2  2
```

Notes

The chunksize parameter is also set in cursor.arraysize object, this parameter can greatly speed up the dump for some databases when the driver uses cursors.arraysize as the number of rows to fetch on a single network trip, but this is driver-dependent, not all drivers implement this (cx_Oracle does it)

See also:

`ploomber.clients.SQLScript`
A task to execute a SQL script and create a table/view as product

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load([key])	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

`build`(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- `TaskBuildError` – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- `DAGBuildEarlyStop` – If any task or on_finish hook raises a DAGBuildEarlyStop error

`debug`()

Debug task, only implemented in certain tasks

`load`(*key=None*, ***kwargs*)

Load task as pandas.DataFrame. Only implemented in certain tasks

`render`(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.SQLTransfer

class ploomber.tasks.**SQLTransfer**(*source*, *product*, *dag*, *name=None*, *client=None*, *params=None*, *chunksize=10000*)

Transfers data from a SQL database to another (Note: this relies on pandas, only use it for small to medium size datasets)

Parameters

- **source** (*str or pathlib.Path*) – SQL script source, if str, the content is interpreted as the actual script, if pathlib.Path, the content of the file is loaded
- **product** (*ploomber.products.product*) – Product generated upon successful execution. For SQLTransfer, usually product.client != task.client. task.client represents the data source while product.client represents the data destination
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag
- **client** (ploomber.clients.SQLAlchemyClient, *optional*) – The client used to connect to the database. Only required if no dag-level client has been declared using dag.clients[class]
- **params** (*dict, optional*) – Parameters to pass to the script, by default, the callable will be executed with a "product" (which will contain the product object). It will also include a "upstream" parameter if the task has upstream dependencies along with any parameters

declared here. The source code is converted to a jinja2.Template for passing parameters, refer to jinja2 documentation for details

- **chunksize** (*int, optional*) – Number of rows to transfer on every chunk

Notes

This task is *not* intended to move large datasets, but a convenience way of transfering small to medium size datasets. It relies on pandas to read and write, which introduces a considerable overhead.

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.SQLUpload

class ploomber.tasks.**SQLUpload**(*source*, *product*, *dag*, *name=None*, *client=None*, *params=None*, *chunksize=None*, *io_handler=None*, *to_sql_kwargs=None*)

Upload data to a SQL database from a parquet or a csv file. Note: this task relies uses pandas.to_sql which introduces some overhead. Only use it for small to medium size datasets. Each database usually come with a tool to upload data efficiently. If you are using PostgreSQL, check out the PostgresCopyFrom task.

Parameters

- **source** (*str or pathlib.Path*) – Path to parquet or a csv file to upload
- **product** (*ploomber.products.product*) – Product generated upon successful execution. The client for the product must be in the target database, where as task.client should be a client in the source database.
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag
- **client** (ploomber.clients.SQLAlchemyClient, *optional*) – The client used to connect to the database and where the data will be uploaded. Only required if no dag-level client has been declared using dag.clients[class]
- **params** (*dict, optional*) – Parameters to pass to the script, by default, the callable will be executed with a "product" (which will contain the product object). It will also include a "upstream" parameter if the task has upstream dependencies along with any parameters

declared here. The source code is converted to a jinja2.Template for passing parameters, refer to jinja2 documentation for details

- **chunksize** (*int, optional*) – Number of rows to transfer on every chunk
- **io_handler** (*callable, optional*) – A Python callable to read the source file, if None, it will tried to be inferred from the source file extension
- **to_sql_kwargs** (*dict, optional*) – Keyword arguments passed to the pandas.DataFrame.to_sql function, one useful parameter is "if_exists", which determines if the task should fail ("fail"), the relation should be replaced ("replace") or rows appended ("append").

Notes

This task is *not* intended to move large datasets, but a convenience way of transfering small to medium size datasets. It relies on pandas to read and write, which introduces a considerable overhead.

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.PostgresCopyFrom

class ploomber.tasks.**PostgresCopyFrom**(*source*, *product*, *dag*, *name=None*, *client=None*, *params=None*, *columns=None*)

Efficiently copy data to a postgres database using COPY FROM (faster alternative to SQLUpload for postgres). If using SQLAlchemy client for postgres is psycopg2. Replaces the table if exists.

Parameters

- **source** (*str or pathlib.Path*) – Path to parquet file to upload
- **client** (ploomber.clients.SQLAlchemyClient, *optional*) – The client used to connect to the database and where the data will be uploaded. Only required if no dag-level client has been declared using dag.clients[class]

Notes

Although this task does not depend on pandas for data i/o, it still needs it to dynamically create the table, after the table is created the COPY statement is used to upload the data

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.ShellScript

class ploomber.tasks.**ShellScript**(*source*, *product*, *dag*, *name=None*, *client=None*, *params=None*)

Execute a shell script.

Parameters

- **source** (*str or pathlib.Path*) – Script source, if str, the content is interpreted as the actual script, if pathlib.Path, the content of the file is loaded. The souce code must have the {{product}} tag
- **product** (ploomber.products.Product) – Product generated upon successful execution
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag
- **client** (ploomber.clients.ShellClient *or RemoteShellClient, optional*) – The client used to connect to the database. Only required if no dag-level client has been declared using dag.clients[class]
- **params** (*dict, optional*) – Parameters to pass to the script, by default, the callable will be executed with a "product" (which will contain the product object). It will also include a "upstream" parameter if the task has upstream dependencies along with any parameters declared here. The source code is converted to a jinja2.Template for passing parameters, refer to jinja2 documentation for details

Examples

Spec API:

See here.

Python API:

```
>>> from          import
>>> from           import
>>> from                 import
>>> from                    import
>>>      = "touch {{product['first']}}; touch {{product['second']}}"
>>>   =      'script.sh' .
>>>     =
>>>         =  'first'       'first.txt'   'second'       'second.txt'
>>>   =                   'script.sh'               =
>>>         =    .
```

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.DownloadFromURL

class ploomber.tasks.**DownloadFromURL**(*source*, *product*, *dag*, *name=None*, *params=None*)

Download a file from a URL (uses urllib.request.urlretrieve)

Parameters

- **source** (*str*) – URL to download the file from
- **product** (ploomber.products.File) – Product generated upon successful execution
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other[, group_name])	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*, *group_name=None*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.Link

class ploomber.tasks.**Link**(*product*, *dag*, *name*)

A dummy Task used to "plug" an external Product to a pipeline, this task is always considered up-to-date

The purpose of this Task is to link a pipeline to an external read-only file, this task does not do anything on the dataset and the product is always considered up-to-date. There are two primary use cases: when the raw data is automatically uploaded to a file (or table) and the pipeline does not have control over data updates, this task can be used to link the pipeline to that file, without having to copy it, downstream tasks will see this dataset as just another Product. The second use case is when developing a prediction pipeline. When making predictions on new data, the pipeline might rely on existing data to generate features, this task can be used to point to such file it can also be used to point to a serialized model, this last scenario is only recommended for prediction pipeline that do not have strict performance requirements, unserializing models is an expensive operation, for real-time predictions, the model should be kept in memory

Parameters

- **product** (ploomber.products.Product) – Product to link to the dag
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other)	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

ploomber.tasks.Input

class ploomber.tasks.**Input**(*product*, *dag*, *name*)

A dummy task used to represent input provided by the user, it is always considered outdated.

When making new predictions, the user must submit some input data to build features and then feed the model, this task can be used to point to such input. It does not perform any processing (read-only data) but it is always considered outdated, which means it will always trigger execution.

Parameters

- **product** (ploomber.products.Product) – Product to to serve as input to the dag
- **dag** (ploomber.DAG) – A DAG to add this task to
- **name** (*str*) – A str to indentify this task. Should not already exist in the dag

Methods

build([force, catch_exceptions])	Build a single task
debug()	Debug task, only implemented in certain tasks
load()	Load task as pandas.DataFrame.
render([force, outdated_by_code, remote])	Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.
run()	This is the only required method Task subclasses must implement
set_upstream(other)	
status([return_code_diff, sections])	Prints the current task status

build(*force=False*, *catch_exceptions=True*)

Build a single task

Although Tasks are primarily designed to execute via DAG.build(), it is possible to do so in isolation. However, this only works if the task does not have any unrendered upstream dependencies, if that's the case, you should call DAG.render() before calling Task.build()

Returns
A dictionary with keys 'run' and 'elapsed'

Return type
dict

Raises

- **TaskBuildError** – If the error failed to build because it has upstream dependencies, the build itself failed or build succeded but on_finish hook failed
- **DAGBuildEarlyStop** – If any task or on_finish hook raises a DAGBuildEarlyStop error

debug()

Debug task, only implemented in certain tasks

load()

Load task as pandas.DataFrame. Only implemented in certain tasks

render(*force=False*, *outdated_by_code=True*, *remote=False*)

Renders code and product, all upstream tasks must have been rendered first, for that reason, this method will usually not be called directly but via DAG.render(), which renders in the right order.

Render fully determines whether a task should run or not.

Parameters

- **force** (*bool, default=False*) – If True, mark status as WaitingExecution/WaitingUpstream even if the task is up-to-date (if there are any File(s) with clients, this also ignores the status of the remote copy), otherwise, the normal process follows and only up-to-date tasks are marked as Skipped.
- **outdated_by_code** (*bool, default=True*) – Factors to determine if Task.product is marked outdated when source code changes. Otherwise just the upstream timestamps are used.
- **remote** (*bool, default=False*) – Use remote metadata to determine status

Notes

This method tries to avoid calls to check for product status whenever possible, since checking product's metadata can be a slow operation (e.g. if metadata is stored in a remote database)

When passing force=True, product's status checking is skipped altogether, this can be useful when we only want to quickly get a rendered DAG object to interact with it

run()

This is the only required method Task subclasses must implement

set_upstream(*other*)

status(*return_code_diff=False*, *sections=None*)

Prints the current task status

Parameters

sections (*list, optional*) – Sections to include. Defaults to "name", "last_run", "outdated", "product", "doc", "location"

Attributes

PRODUCT_CLASSES_ALLOWED	
client	
exec_status	
name	A str that represents the name of the task, you can access tasks in a dag using dag['some_name']
on_failure	Callable to be executed if task fails (passes Task as first parameter and the exception as second parameter)
on_finish	Callable to be executed after this task is built successfully (passes Task as first parameter)
on_render	
params	dict that holds the parameter that will be passed to the task upon execution.
product	The product this task will create upon execution
source	Source is used by the task to compute its output, for most cases this is source code, for example PythonCallable takes a function as source and SQLScript takes a string with SQL code as source.
upstream	A mapping for upstream dependencies {task name} -> [task object]

Products

Product(identifier)	Abstract class for all Products
File(identifier[, client])	A file (or directory) in the local filesystem
SQLRelation(identifier)	A product that represents a SQL relation (table or view) with no metadata (incremental builds won't work).
PostgresRelation(identifier[, client])	A PostgreSQL relation
SQLiteRelation(identifier[, client])	A SQLite relation
GenericSQLRelation(identifier[, client])	A GenericProduct whose identifier is a SQL relation, uses SQLite as metadata backend
GenericProduct(identifier[, client])	GenericProduct is used when there is no specific Product implementation.

ploomber.products.Product

class ploomber.products.**Product**(*identifier*)

Abstract class for all Products

prepare_metadata

A hook to execute before saving metadata, should include a "metadata" parameter and might include "product". "metadata" will be a dictionary with the metadata to save, it is not recommended to change any of the existing keys but additional key-value pairs might be included

Type
callable

Methods

delete([force])	Deletes the product
download()	
exists()	This method returns True if the product exists, it is not part of the metadata, so there is no cached status
fetch_metadata()	
render(params, **kwargs)	Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect
save_metadata(metadata)	
to_json_serializable()	Returns a JSON serializable version of this product
upload()	

abstract delete(*force=False*)

Deletes the product

download()

abstract exists()

This method returns True if the product exists, it is not part of the metadata, so there is no cached status

abstract fetch_metadata()

render(*params*, ***kwargs*)

Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect

abstract save_metadata(*metadata*)

to_json_serializable()

Returns a JSON serializable version of this product

upload()

Attributes

client
task

ploomber.products.File

class ploomber.products.**File**(*identifier*, *client=None*)

A file (or directory) in the local filesystem

Parameters

identifier (*str or pathlib.Path*) – The path to the file (or directory), can contain placeholders (e.g. {{placeholder}})

Methods

delete([force])	Deletes the product
download()	
exists()	This method returns True if the product exists, it is not part of the metadata, so there is no cached status
fetch_metadata()	
render(params, **kwargs)	Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect
save_metadata(metadata)	
to_json_serializable()	Returns a JSON serializable version of this product
upload()	

delete(*force=False*)

Deletes the product

download()

exists()

This method returns True if the product exists, it is not part of the metadata, so there is no cached status

fetch_metadata()

render(*params*, ***kwargs*)

Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect

save_metadata(*metadata*)

to_json_serializable()

Returns a JSON serializable version of this product

upload()

Attributes

client
task

ploomber.products.SQLRelation

class ploomber.products.**SQLRelation**(*identifier*)

A product that represents a SQL relation (table or view) with no metadata (incremental builds won't work). See *ploomber.products.GenericSQLRelation* if you want to enable incremental builds.

Parameters

identifier (*tuple of length 3*) – A tuple with (schema, name, kind) where kind must be either 'table' or 'view'

See also:

ploomber.products.GenericSQLRelation

SQL relation (table or view) that stores metadata (to enable incremental builds) in a SQLite database.

Methods

delete([force])	Deletes the product
download()	
exists()	This method returns True if the product exists, it is not part of the metadata, so there is no cached status
fetch_metadata()	
render(params, **kwargs)	Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect
save_metadata(metadata)	
to_json_serializable()	Returns a JSON serializable version of this product
upload()	

delete(*force=False*)

Deletes the product

download()

exists()

This method returns True if the product exists, it is not part of the metadata, so there is no cached status

fetch_metadata()

render(*params*, ***kwargs*)

Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect

save_metadata(*metadata*)

to_json_serializable()

Returns a JSON serializable version of this product

upload()

Attributes

client
kind
name
schema
task

ploomber.products.PostgresRelation

class ploomber.products.**PostgresRelation**(*identifier*, *client=None*)

A PostgreSQL relation

Parameters

- **identifier** (*tuple of length 3*) – A tuple with (schema, name, kind) where kind must be either 'table' or 'view'
- **client** (ploomber.clients.DBAPIClient *or* SQLAlchemyClient, *optional*) – The client used to connect to the database. Only required if no dag-level client has been declared using dag.clients[class]

Examples

```
>>> from                    import
>>>          =                    'schema'  'some_table'  'table'
>>>               # returns qualified name
'schema.some_table'
```

Methods

delete([force])	Deletes the product
download()	
exists()	This method returns True if the product exists, it is not part of the metadata, so there is no cached status
fetch_metadata()	
render(params, **kwargs)	Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect
save_metadata(metadata)	
to_json_serializable()	Returns a JSON serializable version of this product
upload()	

delete(*force=False*)

Deletes the product

download()

exists()

This method returns True if the product exists, it is not part of the metadata, so there is no cached status

fetch_metadata()

render(*params*, ***kwargs*)

Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect

save_metadata(*metadata*)

to_json_serializable()

Returns a JSON serializable version of this product

upload()

Attributes

client
kind
name
schema
task

ploomber.products.SQLiteRelation

class ploomber.products.**SQLiteRelation**(*identifier*, *client=None*)

A SQLite relation

Parameters

- **identifier** (*tuple of length 3 or 2*) – A tuple with (schema, name, kind) where kind must be either 'table' or 'view'. If passed a tuple with length 2, schema is assumed None. Schemas in SQLite represent other databases when using the ATTACH command.
- **client** (ploomber.clients.DBAPIClient *or* SQLAlchemyClient, *optional*) – The client used to connect to the database. Only required if no dag-level client has been declared using dag.clients[class]

Examples

```
>>> from                    import
>>>          =                  'schema'  'some_table'  'table'
>>>                # returns qualified name
'schema.some_table'
```

Methods

delete()	Deletes the product
download()	
exists()	This method returns True if the product exists, it is not part of the metadata, so there is no cached status
fetch_metadata()	
render(params, **kwargs)	Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect
save_metadata(metadata)	
to_json_serializable()	Returns a JSON serializable version of this product
upload()	

delete()

Deletes the product

download()

exists()

This method returns True if the product exists, it is not part of the metadata, so there is no cached status

fetch_metadata()

render(*params*, ***kwargs*)

Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect

save_metadata(*metadata*)

to_json_serializable()

Returns a JSON serializable version of this product

upload()

Attributes

client	
kind	
name	Used as identifier in the database
schema	
task	

ploomber.products.GenericSQLRelation

class ploomber.products.**GenericSQLRelation**(*identifier*, *client=None*)

A GenericProduct whose identifier is a SQL relation, uses SQLite as metadata backend

Parameters

- **identifier** (*tuple of length 3*) – A tuple with (schema, name, kind) where kind must be either 'table' or 'view'
- **client** (ploomber.clients.DBAPIClient *or* SQLAlchemyClient, *optional*) – The client used to *store metadata for this product*. Only required if no dag-level client has been declared using dag.clients[class]

See also:

ploomber.products.SQRelation
SQL relation (table or view) with no metadata.

Methods

delete([force])	Deletes the product
download()	
exists()	This method returns True if the product exists, it is not part of the metadata, so there is no cached status
fetch_metadata()	
render(params, **kwargs)	Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect
save_metadata(metadata)	
to_json_serializable()	Returns a JSON serializable version of this product
upload()	

delete(*force=False*)

Deletes the product

download()

exists()

This method returns True if the product exists, it is not part of the metadata, so there is no cached status

fetch_metadata()

render(*params*, ***kwargs*)

Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect

save_metadata(*metadata*)

to_json_serializable()

Returns a JSON serializable version of this product

upload()

Attributes

client	
kind	
name	Used as identifier in the database
schema	
task	

ploomber.products.GenericProduct

class ploomber.products.**GenericProduct**(*identifier*, *client=None*)

GenericProduct is used when there is no specific Product implementation. Sometimes it is technically possible to write a Product implementation but if you don't want to do it you can use this one. Other times is is not possible to provide a concrete Product implementation (e.g. we cannot store arbitrary metadata in a Hive table). GenericProduct works as any other product but its metadata is stored not in the Product itself but in a different backend.

Parameters

- **identifier** (*str*) – An identifier fot this product, can contain placeholders (e.g. {{placeholder}})
- **client** (ploomber.clients.DBAPIClient *or* SQLAlchemyClient, *optional*) – The client used to *store metadata for this product*. Only required if no dag-level client has been declared using dag.clients[class]

Notes

exists does not check for product existence, just checks if metadata exists delete does not perform actual deletion, just deletes metadata

Methods

delete([force])	Deletes the product
download()	
exists()	This method returns True if the product exists, it is not part of the metadata, so there is no cached status
fetch_metadata()	
render(params, **kwargs)	Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect
save_metadata(metadata)	
to_json_serializable()	Returns a JSON serializable version of this product
upload()	

delete(*force=False*)

Deletes the product

download()

exists()

This method returns True if the product exists, it is not part of the metadata, so there is no cached status

fetch_metadata()

render(*params*, ***kwargs*)

Render Product - this will render contents of Templates used as identifier for this Product, if a regular string was passed, this method has no effect

save_metadata(*metadata*)

to_json_serializable()

Returns a JSON serializable version of this product

upload()

Attributes

client	
name	Used as identifier in the database
task	

Clients

Client()	Abstract class for all clients
DBAPIClient(connect_fn, connect_kwargs[, ...])	A client for a PEP 249 compliant client library
SQLAlchemyClient(uri[, split_source, ...])	Client for connecting with any SQLAlchemy supported database
ShellClient([run_template, ...])	Client to run command in the local shell
S3Client(bucket_name, parent[, ...])	Client for uploading File products to Amazon S3
GCloudStorageClient(bucket_name, parent[, ...])	Client for uploading File products to Google Cloud Storage

ploomber.clients.Client

class ploomber.clients.**Client**

Abstract class for all clients

Clients are classes that communicate with another system (usually a database), they provide a thin wrapper around libraries that implement clients to avoid managing connections directly. The most common use case by far is for a Task/Product to submit some code to a system, a client just provides a way of doing so without dealing with connection details.

A Client is reponsible for making sure an open connection is available at any point (open a connection if none is available).

However, clients are not strictly necessary, a Task/Product could manage their own client connections. For example the NotebookRunner task does have a Client since it only calls an external library to run.

Notes

Method's names were chosen to resemble the ones in the Python DB API Spec 2.0 (PEP 249)

Methods

close()	Close connection if there is one active
execute(code)	Execute code

abstract close()

Close connection if there is one active

abstract execute(*code*)

Execute code

Attributes

connection	Return a connection, open one if there isn't any

ploomber.clients.DBAPIClient

class ploomber.clients.**DBAPIClient**(*connect_fn*, *connect_kwargs*, *split_source=None*)

A client for a PEP 249 compliant client library

Parameters

- **connect_fn** (*callable*) – The function to use to open the connection
- **connect_kwargs** (*dict*) – Keyword arguments to pass to connect_fn
- **split_source** (*str, optional*) – Some database drivers do not support multiple commands in a single execute statement. Use this optiion to split commands by a given character (e.g. ';') and send them one at a time. Defaults to None (no splitting)

Examples

Spec API:

Given the following `clients.py`:

```
import
from                    import

def get
    return                      .                       ='my.db'
```

Spec API (dag-level client):

```
clients
    # key is a task class such as SQLDump or SQLScript
    SQLDump  clients.get

tasks
      source  query.sql
      product  output/data.csv
```

Spec API (task-level client):

```
tasks
      source  query.sql
      product  output/data.csv
      client  clients.get
```

Python API (dag-level client):

```
>>> import
>>> import        as
>>> from          import
>>> from                  import
>>> from                import
>>> from                  import
>>>          =        .               ='my.db'
>>>    =    .          'a'        100   'b'        100
>>>    =    .       'numbers'               =False
>>>         .
>>>      =
>>>         =                    .                       ='my.db'
>>>    .                 =        # dag-level client
>>>    =          'SELECT * FROM numbers'       'data.parquet'
...                  =         ='dump'
...                     =
...                        =None  # no need to pass client here
>>>    =    .
>>>     =   .             'data.parquet'
>>>    .     3
   a  b
0  0  0
1  1  1
2  2  2
```

Python API (task-level client):

```
>>> import
>>> import        as
>>> from          import
>>> from                  import
>>> from                import
>>> from                  import
>>>          =        .               ='some.db'
>>>    =    .          'a'        100   'b'        100
>>>    =    .       'numbers'               =False
```

(continues on next page)

(continued from previous page)

```
>>>         .
>>>      =
>>>         =                        .                          ='some.db'
>>>    =          'SELECT * FROM numbers'        'data.parquet'
...                   =          ='dump'
...                       =           # pass client to task
...                          =None
>>>    =     .
>>>     =    .              'data.parquet'
>>>    .      3
   a  b
0  0  0
1  1  1
2  2  2
```

See also:

ploomber.clients.SQLAlchemyClient
A client to connect to a database using sqlalchemy as backend

Methods

close()	Close connection if there is an active one
cursor()	
execute(code)	Execute code with the existing connection

close()
Close connection if there is an active one

cursor()

execute(*code*)
Execute code with the existing connection

Attributes

connection	Return a connection, open one if there isn't any

ploomber.clients.SQLAlchemyClient

class ploomber.clients.**SQLAlchemyClient**(*uri*, *split_source='default'*, *create_engine_kwargs=None*)

Client for connecting with any SQLAlchemy supported database

Parameters

- **uri** (*str or sqlalchemy.engine.url.URL*) – URI to pass to sqlalchemy.create_engine or URL object created using sqlalchemy.engine.url.URL.create
- **split_source** (*str, optional*) – Some database drivers do not support multiple commands in a single execute statement. Use this option to split commands by a given character (e.g. ';') and send them one at a time. Defaults to 'default', which splits by ';' if using SQLite database, but does not perform any splitting with other databases. If None, it will never split, a string value is interpreted as the token to use for splitting statements regardless of the database type
- **create_engine_kwargs** (*dict, optional*) – Keyword arguments to pass to sqlalchemy.create_engine

Notes

SQLite client does not support sending more than one command at a time, if using such backend code will be split and several calls to the db will be performed.

Examples

Spec API:

Given the following `clients.py`:

```
import
from                  import

def get
        =           .      .   .   .                ='sqlite'
                                                   ='my_db.db'
    return
```

Spec API (dag-level client):

```
clients
    # key is a task class such as SQLDump or SQLScript
    SQLDump  clients.get

tasks
      source  query.sql
      product  output/data.csv
```

Spec API (task-level client):

```
tasks
      source  query.sql
      product  output/data.csv
      client  clients.get
```

Python API (dag-level client):

```
>>> import
>>> import
>>> import        as
>>> from          import
>>> from                  import
>>> from               import
>>> from                 import
>>>         =        .               ='my.db'
>>>    =    .          'a'        100   'b'        100
>>>   =    .       'numbers'               =False
>>>        .
>>>     =
>>>     =           .      .   .   .                ='sqlite'
...                                                ='my.db'
>>>        =
>>>    .                =        # dag-level client
>>>   =         'SELECT * FROM numbers'       'data.parquet'
...                =         ='dump'
...                       =None  # no need to pass client here
>>>   =    .
>>>    =    .             'data.parquet'
>>>   .     3
   a  b
0  0  0
1  1  1
2  2  2
```

Python API (task-level client):

```
>>> import
>>> import
>>> import        as
>>> from          import
>>> from                  import
>>> from               import
>>> from                 import
>>>         =        .               ='some.db'
>>>    =    .          'a'        100   'b'        100
>>>   =    .       'numbers'               =False
>>>        .
>>>     =
>>>     =           .      .   .   .                ='sqlite'
...                                                ='some.db'
>>>        =
>>>   =         'SELECT * FROM numbers'       'data.parquet'
...                =         ='dump'
...                   =        # pass client to task
...                      =None
>>>   =    .
>>>    =    .             'data.parquet'
>>>   .     3
   a  b
```

(continues on next page)

(continued from previous page)

```
0  0  0
1  1  1
2  2  2
```

See also:

ploomber.clients.DBAPIClient
A client to connect to a database

Methods

close()	Closes all connections
cursor()	
execute(code)	Execute code

close()
Closes all connections

cursor()

execute(*code*)
Execute code

Attributes

connection	Return a connection from the pool
engine	Returns a SQLAlchemy engine
split_source_mapping	

ploomber.clients.ShellClient

class ploomber.clients.**ShellClient**(*run_template='bash {{path_to_code}}'*, *subprocess_run_kwargs={'shell': False, 'stderr': -1, 'stdout': -1}*)

Client to run command in the local shell

Parameters

- **run_template** (*str*) – Template for executing commands, must include the {{path_to_code}} placeholder which will point to the rendered code. Defaults to 'bash {{path_to_code}}'
- **subprocess_run_kwargs** (*dict*) – Keyword arguments to pass to the subprocess.run when running run_template

Methods

close()	Close connection if there is one active
execute(code)	Run code

close()

Close connection if there is one active

execute(*code*)

Run code

Attributes

connection	Return a connection, open one if there isn't any

ploomber.clients.S3Client

class ploomber.clients.**S3Client**(*bucket_name*, *parent*, *json_credentials_path=None*, *path_to_project_root=None*, *credentials_relative_to_project_root=True*, ***kwargs*)

Client for uploading File products to Amazon S3

Parameters

- **bucket_name** (*str*) – Bucket to use
- **parent** (*str*) – Parent folder in the bucket to store files. For example, if `parent='path/to'`, and a product in your pipeline is `out/data.csv`, your file will appear in the bucket at `path/to/out/data.csv`.
- **json_credentials_path** (*str, default=None*) – JSON file to authenticate the client. Must contain `aws_access_key_id` and `aws_secret_access_key`. If None, client is initialized without arguments (i.e., `boto3.client('s3')`)
- **path_to_project_root** (*str, default=None*) – Path to project root. If None, looks it up automatically and assigns it to the parent folder of your `pipeline.yaml` spec or `setup.py` (if your project is a package). This determines the path in remote storage. For example, if `path_to_project_root` is `/my-project`, you're storing a product at `/my-project/out/data.csv`, and `parent='some-dir'`, the file will be stored in the bucket at `some-dir/out/data.csv` (we first compute the path of your product relative to the project root, then prefix it with `parent`).
- **credentials_relative_to_project_root** (*bool, default=True*) – If `True`, relative paths in `json_credentials_path` are so to the `path_to_project_root`, instead of the current working directory
- ****kwargs** – Keyword arguments for the client constructor

Examples

Spec API:

Given the following `clients.py`:

```
import
from                  import

def get
    return                    ='my-bucket'        ='my-pipeline'
```

Spec API (dag-level client):

```
clients
    # all files from all tasks will be uploaded
    File  clients.get

tasks
      source  notebook.ipynb
      product  output/report.html
```

Spec API (dag-level client, custom arguments):

```
clients
    # if your get function takes arguments, pass them like this
    File
        dotted_path  clients.get
        arg  value
        ...

tasks
      source  notebook.ipynb
      product  output/report.html
```

Spec API (product-level client):

```
tasks
      source  notebook.ipynb
      product_client  clients.get
      # outputs from this task will be uploaded
      product  output/report.html
```

Python API (dag-level client):

```
>>> from          import
>>> from                   import
>>> from                 import
>>> from                   import
>>>     =
>>>        =                    ='my-bucket'        ='my-pipeline'
...                                        ='.'
>>>    .              =        # dag-level client
>>>     =
>>> def my_function
```

(continues on next page)

(continued from previous page)

```
...                 .
>>>   =                                 'file.txt'      =
>>>    .
```

Python API (product-level client):

```
>>> from          import
>>> from                  import
>>> from               import
>>> from                 import
>>>     =
>>>        =                   ='my-bucket'        ='my-pipeline'
...                                        ='.'
>>>     =
>>> def my_function
...                   .
>>>         =      'file.txt'        =
>>>   =                                          =
>>>    .
```

See also:

ploomber.clients.GCloudStorageClient
Client for uploading products to Google Cloud Storage

Notes

If a notebook (or script) task fails, the partially executed `.ipynb` file will be uploaded using this client.

Methods

close()	
download(local[, destination])	Download remote copy of a given local path.
upload(local)	Upload file or folder from a local path by calling _upload as needed

close()

download(*local*, *destination=None*)
Download remote copy of a given local path. Local may be a file or a folder (all contents downloaded).

Parameters

- **local** – Path to local file or folder whose remote copy will be downloaded
- **destination** – Download location. If None, overwrites local copy

upload(*local*)
Upload file or folder from a local path by calling _upload as needed

Parameters
local – Path to local file or folder to upload

Attributes

parent	Parent where all products are stored

ploomber.clients.GCloudStorageClient

class ploomber.clients.**GCloudStorageClient**(*bucket_name*, *parent*, *json_credentials_path=None*, *path_to_project_root=None*, *credentials_relative_to_project_root=True*, ***kwargs*)

Client for uploading File products to Google Cloud Storage

Parameters

- **bucket_name** (*str*) – Bucket to use
- **parent** (*str*) – Parent folder in the bucket to store files. For example, if `parent='path/to'`, and a product in your pipeline is `out/data.csv`, your file will appea in the bucket at `path/to/out/data.csv`.
- **json_credentials_path** (*str, default=None*) – Use the given JSON file to authenticate the client (uses `Client.from_service_account_json(**kwargs)`), if `None`, initializes the client using `Client(**kwargs)`
- **path_to_project_root** (*str, default=None*) – Path to project root. If None, looks it up automatically and assigns it to the parent folder of your `pipeline.yaml` spec or `setup.py` (if your project is a package). This determines the path in remote storage. For example, if `path_to_project_root` is `/my-project`, you're storing a product at `/my-project/out/data.csv`, and `parent='some-dir'`, the file will be stored in the bucket at `some-dir/out/data.csv` (we first compute the path of your product relative to the project root, then prefix it with `parent`).
- **credentials_relative_to_project_root** (*bool, default=True*) – If `True`, relative paths in `json_credentials_path` are so to the `path_to_project_root`, instead of the current working directory
- ****kwargs** – Keyword arguments for the client constructor

Examples

Spec API:

Given the following `clients.py`:

```
import
from                    import

def get
    return                                  ='my-bucket'
                                          ='my-pipeline'
```

Spec API (dag-level client):

```
clients
    # all files from all tasks will be uploaded
    File  clients.get

tasks
      source  notebook.ipynb
      product  output/report.html
```

Spec API (dag-level client, custom arguments):

```
clients
    # if your get function takes arguments, pass them like this
    File
        dotted_path  clients.get
        arg  value
        ...

tasks
      source  notebook.ipynb
      product  output/report.html
```

Spec API (product-level client):

```
tasks
      source  notebook.ipynb
      product_client  clients.get
      # outputs from this task will be uploaded
      product  output/report.html
```

Python API (dag-level client):

```
>>> from          import
>>> from                   import
>>> from                 import
>>> from                   import
>>>     =
>>>        =                                ='my-bucket'
...                                    ='my-pipeline'
...                                                 ='.'
>>>    .              =        # dag-level client
>>>     =
>>> def my_function
...                   .
>>>    =                                  'file.txt'      =
>>>     .
```

Python API (product-level client):

```
>>> from          import
>>> from                   import
>>> from                 import
>>> from                   import
>>>     =
```

(continues on next page)

(continued from previous page)

```
>>>         =                                    ='my-bucket'
...                                          ='my-pipeline'
...                                                      ='.'
>>>      =
>>> def my_function
...                     .
>>>          =       'file.txt'         =
>>>    =                                              =
>>>     .
```

See also:

ploomber.clients.S3Client
Client for uploading products to Amazon S3

Notes

Complete example using the Spec API

If a notebook (or script) task fails, the partially executed `.ipynb` file will be uploaded using this client.

Methods

close()	
download(local[, destination])	Download remote copy of a given local path.
upload(local)	Upload file or folder from a local path by calling _upload as needed

close()

download(*local*, *destination=None*)

Download remote copy of a given local path. Local may be a file or a folder (all contents downloaded).

Parameters

- **local** – Path to local file or folder whose remote copy will be downloaded
- **destination** – Download location. If None, overwrites local copy

upload(*local*)

Upload file or folder from a local path by calling _upload as needed

Parameters

local – Path to local file or folder to upload

Attributes

parent	Parent where all products are stored

Spec

DAGSpec(data[, env, lazy_import, reload, ...])	A DAG spec is a dictionary with certain structure that can be converted to a DAG using DAGSpec.to_dag().

ploomber.spec.DAGSpec

class ploomber.spec.**DAGSpec**(*data*, *env=None*, *lazy_import=False*, *reload=False*, *parent_path=None*)

A DAG spec is a dictionary with certain structure that can be converted to a DAG using `DAGSpec.to_dag()`.

There are two cases: the simplest one is just a dictionary with a "location" key with the factory to call, the other explicitly describes the DAG structure as a dictionary.

When `.to_dag()` is called, the current working directory is temporarily switched to the spec file parent folder (only applies when loading from a file)

Parameters

- **data** (*str, pathlib.Path or dict*) – Path to a YAML spec or dict spec. If loading from a file, sources and products are resolved to the file's parent. If the file is in a packaged structure (i.e., src/package/pipeline.yaml), the existence of a setup.py in the same folder as src/ is validated. If loaded from a dict, sources and products aren't resolved, unless a parent_path is not None.
- **env** (*dict, pathlib.path or str, optional*) – If path it must be a YAML file. Environment to load. Any string with the format '{{placeholder}}' in the spec is replaced by the corresponding value in the given key (i.e., placeholder). If `env` is None and spec is a dict, no env is loaded. If None and loaded from a YAML file, an `env.yaml` file is loaded from the current working diirectory, if it doesn't exist, it is loaded from the YAML spec parent folder. If none of these exist, no env is loaded. A *ploomber.Env* object is initialized, see documentation for details.
- **lazy_import** (*bool, optional*) – Whether to import dotted paths to initialize PythonCallables with the actual function. If False, PythonCallables are initialized directly with the dotted path, which means some verifications such as import statements in that function's module are delayed until the pipeline is executed. This also applies to placeholders loaded using a SourceLoader, if a template exists, it will return the path to it, instead of initializing it, if it doesn't, it will return None instead of raising an error. This setting is useful when we require to load YAML spec and instantiate the DAG object to extract information from it (e.g., which are the declared tasks) but the process running it may not have all the required dependencies to do so (e.g., an imported library in a PythonCallable task).
- **reload** (*bool, optional*) – Reloads modules before importing dotted paths to detect code changes if the module has already been imported. Has no effect if lazy_import=True.

Examples

Load from `pipeline.yaml`:

```
>>> from               import
>>>      =          'spec/pipeline.yaml'  # load spec
>>>     =     .          # convert to DAG
>>>        =    .
```

Override `env.yaml`:

```
>>> from               import
>>>      =          'spec/pipeline.yaml'     =        ='value'
>>>     =     .
>>>        =    .
```

See also:

ploomber.DAG
: Pipeline internal representation, implements the methods in the command-line interface (e.g., `DAG.build()`, or `DAG.plot`)

path
: Returns the path used to load the data. None if loaded from a dictionary

  Type
  : str or None

Methods

clear()	
find([env, reload, lazy_import, ...])	Automatically find pipeline.yaml and return a DAGSpec object, which can be converted to a DAG using .to_dag()
from_directory(path_to_dir)	Construct a DAGSpec from a directory.
from_files(files)	Construct DAGSpec from list of files or glob-like pattern.
get(k[,d])	
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
to_dag()	Converts the DAG spec to a DAG object
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

clear() → None. Remove all items from D.

classmethod find(*env=None*, *reload=False*, *lazy_import=False*, *starting_dir=None*, *name=None*)

Automatically find pipeline.yaml and return a DAGSpec object, which can be converted to a DAG using .to_dag()

Parameters

- **env** – The environment to pass to the spec
- **name** (*str, default=None*) – Filename to search for. If None, it looks for a pipeline.yaml file, otherwise it looks for a file with such name.

classmethod from_directory(*path_to_dir*)

Construct a DAGSpec from a directory. Product and upstream are extracted from sources

Parameters

path_to_dir (*str*) – The directory to use. Looks for scripts (`.py`, `.R` or `.ipynb`) in the directory and interprets them as task sources, file names are assigned as task names (without extension). The spec is generated with the default values in the "meta" section. Ignores files with invalid extensions.

Notes

`env` is not supported because the spec is generated from files in `path_to_dir`, hence, there is no way to embed tags

classmethod from_files(*files*)

Construct DAGSpec from list of files or glob-like pattern. Product and upstream are extracted from sources

Parameters
files (*list or str*) – List of files to use or glob-like string pattern. If glob-like pattern, ignores directories that match the criteria.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

to_dag()
Converts the DAG spec to a DAG object

update([*E*,]**F) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

Attributes

path

Env

with_env(source)	A function decorated with @with_env that starts and environment during the execution of a function.
load_env(fn)	A function decorated with @load_env will be called with the current environment in an env keyword argument
Env([source])	Return the current environment

ploomber.with_env

ploomber.**with_env**(*source*)

A function decorated with @with_env that starts and environment during the execution of a function.

Notes

The first argument of a function decorated with @with_env must be named "env", the env will be passed automatically when calling the function. The original function's signature is edited.

You can replace values in the environment, e.g. if you want to replace env.key.another, you can call the decorated function with: my_fn(env__key__another='my_new_value')

The environment is resolved at import time, changes to the working directory will not affect initialization.

Examples

ploomber.load_env

ploomber.**load_env**(*fn*)

A function decorated with @load_env will be called with the current environment in an env keyword argument

ploomber.Env

class ploomber.**Env**(*source=None*)

Return the current environment

NOTE: this API is experimental and subject to change, it is recommmended to use @with_env and @load_env decorators instead

Env provides a clean and consistent way of managing environment and configuration settings. Its simplest usage provides access to settings specified via an *env.yaml*.

Settings managed by Env are intended to be runtime constant (they are NOT intended to be used as global variables). For example you might want to store database URIs. Storing sensitive information is discouraged as yaml files are plain text. Use *keyring* for that instead.

All sections are optional, but if there is a path section, all values inside that section will be casted to pathlib.Path objects, expanduser() is applied so "~" can be used. Strings with a trailing "/" will be interpreted as directories and they will be created if they do not exist

There are a few placeholders available: * {{user}} expands to the current user (by calling getpass.getuser()) * {{version}} expands to module.__version__ if _module is defined * {{git}} expands to branch name if at the tip, otherwise to the current commit hash (_module has to be defined)

Examples

```
>>> from           import
>>>     =      'db'    'uri'   'my_uri'    'path'    'raw'   '/path/to/raw'
>>>     .   .
'my_uri'
>>>     .     .
PosixPath('/path/to/raw')
```

Notes

Envs are intended to be short-lived, the recommended usage is to start and end them only during the execution of a function that builds a DAG by using the @with_env and @load_env decorators

Methods

end()	End environment.
load()	

classmethod end()

End environment. Usage is discouraged, a single environment is expected to exist during the entire Python process lifespan to avoid inconsistencies, use it only if you have a very strong reason to

classmethod load()

Attributes

name

Serialization

serializer([extension_mapping, fallback, ...])	Decorator for serializing functions
serializer_pickle(obj, product)	A serializer that pickles everything
unserializer([extension_mapping, fallback, ...])	Decorator for unserializing functions
unserializer_pickle(product)	An unserializer that unpickles everything

ploomber.io.serializer

ploomber.io.**serializer**(*extension_mapping=None*, *, *fallback=False*, *defaults=None*, *unpack=False*)

Decorator for serializing functions

Parameters

- **extension_mapping** (*dict, default=None*) – An extension -> function mapping. Calling the decorated function with a File of a given extension will use the one in the mapping if it exists, e.g., {'.csv': to_csv, '.json': to_json}.
- **fallback** (*bool or str, default=False*) – Determines what method to use if extension_mapping does not match the product to serialize. Valid values are True (uses the pickle module), 'joblib', and 'cloudpickle'. If you use any of the last two, the corresponding moduel must be installed. If this is enabled, the body of the decorated function is never executed. To turn it off pass False.
- **defaults** (*list, default=None*) – Built-in serializing functions to use. Must be a list with any combinations of values: '.txt', '.json', '.csv', '.parquet'. To save to .txt, the returned object must be a string, for .json it must be a json serializable object (e.g., a list or a dict), for .csv and .parquet it must be a pandas.DataFrame. If using .parquet, a parquet library must be installed (e.g., pyarrow). If extension_mapping and defaults contain overlapping keys, an error is raised
- **unpack** (*bool, default=False*) – If True, it treats every element in a dictionary as a different file, calling the serializing function one per (key, value) pair and using the key as filename.

ploomber.io.serializer_pickle

ploomber.io.**serializer_pickle**(*obj*, *product*)

A serializer that pickles everything

ploomber.io.unserializer

ploomber.io.**unserializer**(*extension_mapping=None*, *, *fallback=False*, *defaults=None*, *unpack=False*)

Decorator for unserializing functions

Parameters

- **extension_mapping** (*dict, default=None*) – An extension -> function mapping. Calling the decorated function with a File of a given extension will use the one in the mapping if it exists, e.g., {'.csv': from_csv, '.json': from_json}.
- **fallback** (*bool or str, default=False*) – Determines what method to use if extension_mapping does not match the product to unserialize. Valid values are True (uses the pickle module), 'joblib', and 'cloudpickle'. If you use any of the last two, the corresponding moduel must be installed. If this is enabled, the body of the decorated function is never executed. To turn it off pass False.
- **defaults** (*list, default=None*) – Built-in unserializing functions to use. Must be a list with any combinations of values: '.txt', '.json', '.csv', '.parquet'. Unserializing .txt, returns a string, for .json returns any JSON-unserializable object (e.g., a list or a dict), .csv and .parquet return a pandas.DataFrame. If using .parquet, a parquet library must be installed (e.g., pyarrow). If extension_mapping and defaults contain overlapping keys, an error is raises

- **unpack** (*bool, default=False*) – If True and the task product points to a directory, it will call the unserializer one time per file in the directory. The unserialized object will be a dictionary where keys are the filenames and values are the unserialized objects. Note that this isn't recursive, it only looks at files that are immediate children of the product directory.

ploomber.io.unserializer_pickle

ploomber.io.**unserializer_pickle**(*product*)

An unserializer that unpickles everything

Executors

Serial([build_in_subprocess, ...])	Executor than runs one task at a time
Parallel([processes, print_progress, ...])	Runs a DAG in parallel using multiprocessing

ploomber.executors.Serial

class ploomber.executors.**Serial**(*build_in_subprocess=True*, *catch_exceptions=True*, *catch_warnings=True*)

Executor than runs one task at a time

Parameters

- **build_in_subprocess** (*bool, optional*) – Determines whether tasks should be executed in a subprocess or in the current process. For pipelines with a lot of PythonCallables loading large objects such as pandas.DataFrame, this option is recommended as it guarantees that memory will be cleared up upon task execution. Defaults to True.
- **catch_exceptions** (*bool, optional*) – Whether to catch exceptions raised when building tasks and running hooks. If True, exceptions are collected and displayed at the end, downstream tasks of failed ones are aborted (not executed at all), If any task raises a DAGBuildEarlyStop exception, the final exception raised will be of such type. If False, no catching is done, on_failure won't be executed and task status will not be updated and tracebacks from build and hooks won't be logged. Setting of to False is only useful for debugging purposes.
- **catch_warnings** (*bool, optional*) – If True, the executor catches all warnings raised by tasks and displays them at the end of execution. If catch_exceptions is True and there is an error building the DAG, capture warnings are still shown before raising the collected exceptions.

Examples

Spec API:

```
# add at the top of your pipeline.yaml
executor  serial

tasks
    source  script.py
    nb_product_key
```

(continues on next page)

(continued from previous page)

```
    product
        nb_ipynb  nb.ipynb
        nb_html  report.html
```

Python API:

```
>>> from          import
>>> from                    import
>>>     =             ='parallel'  # use with default values
>>>     =             =                          =False   # customize
```

DAG can exit gracefully on function tasks (PythonCallable):

```
>>> from                   import
>>> from                import
>>> from                     import
>>> # A PythonCallable function that raises DAGBuildEarlyStop
>>> def early_stop_root
...     raise                   'Ending gracefully'
```

```
>>> # Since DAGBuildEarlyStop is raised, DAG will exit gracefully.
>>>     =             ='parallel'
>>>   =                                    'file.txt'      =
>>>    .
```

DAG can also exit gracefully on notebook tasks:

```
>>> from         import
>>> from                import
>>> from                   import
>>> def early_stop
...     raise                   'Ending gracefully'
```

```
>>> # Use task-level hook "on_finish" to exit DAG gracefully.
>>>     =             ='parallel'
>>>   =                    'nb.ipynb'        'report.html'      =
>>>  .          =
>>>    .
```

See also:

ploomber.executors.Parallel
Parallel executor

Methods

ploomber.executors.Parallel

class ploomber.executors.**Parallel**(*processes=None*, *print_progress=False*, *start_method=None*)

Runs a DAG in parallel using multiprocessing

Parameters

- **processes** (*int, default=None*) – The number of processes to use. If None, uses os.cpu_count
- **print_progress** (*bool, default=False*) – Whether to print progress to stdout, otherwise just log it
- **start_method** (*str, default=None*) – The method which should be used to start child processes. method can be 'fork', 'spawn' or 'forkserver'. If None or empty then the default start_method is used.

Examples

Spec API:

```
# add at the top of your pipeline.yaml
executor  parallel

tasks
    source  script.py
    nb_product_key
    product
        nb_ipynb  nb.ipynb
        nb_html  report.html
```

Python API:

```
>>> from          import
>>> from                    import
>>>     =            ='parallel'  # use with default values
>>>     =            =                 =2   # customize
```

DAG can exit gracefully on function tasks (PythonCallable):

```
>>> from                   import
>>> from                import
>>> from                     import
>>> # A PythonCallable function that raises DAGBuildEarlyStop
>>> def early_stop_root
...     raise                   'Ending gracefully'
```

```
>>> # Since DAGBuildEarlyStop is raised, DAG will exit gracefully.
>>>     =             ='parallel'
>>>   =                                    'file.txt'      =
>>>    .
```

DAG can also exit gracefully on notebook tasks:

```
>>> from         import
>>> from               import
>>> from                  import
>>> def early_stop
...     raise                  'Ending gracefully'
```

```
>>> # Use task-level hook "on_finish" to exit DAG gracefully.
>>>     =             ='parallel'
>>>   =                    'nb.ipynb'        'report.html'      =
>>>  .          =
>>>    .
```

Notes

If any task crashes, downstream tasks execution is aborted, building continues until no more tasks can be executed

New in version 0.20: Added *start_method* argument

See also:

ploomber.executors.Serial
Serial executor

Methods

Attributes

multiprocessing_start_methods

SourceLoader

SourceLoader([path, module])	Load source files using a jinja2.Environment

ploomber.SourceLoader

class ploomber.**SourceLoader**(*path=None*, *module=None*)

Load source files using a jinja2.Environment

Data pipelines usually rely on non-Python source code such as SQL scripts, SourceLoader provides a convenient way of loading them. This serves two purposes: 1) Avoid hardcoded paths to files and 2) Allows using advanced jinja2 features that require an Environment object such as macros (under the hood, SourceLoader initializes an Environment with a FileSystemLoader) SourceLoader returns ploomber.Placeholder objects that can be directly passed to Tasks in the source parameter

Parameters

- **path** (*str, pathlib.Path, optional*) – Path (or list of) to load files from. Required if module is None
- **module** (*str or module, optional*) – Module name as dotted string or module object. Preprends to path parameter

Examples

```
>>> from          import
>>>        =              'path/to/templates/'
>>>        'load_customers.sql'
>>>       .             'load_customers.sql'
```

Methods

get(key)	Load template, returns None if it doesn' exist
get_template(name)	Load a template by name
path_to(key)	Return the path to a template, even if it doesn't exist

get(*key*)

Load template, returns None if it doesn' exist

get_template(*name*)

Load a template by name

Parameters

name (*str or pathlib.Path*) – Template to load

path_to(*key*)

Return the path to a template, even if it doesn't exist

2.6.4 Testing utilities

SQL

nulls_in_columns(client, cols, product)	Check if any column has NULL values, returns bool
distinct_values_in_column(client, col, product)	Get distinct values in a column
duplicates_in_column(client, col, product)	Check if a column (or group of columns) has duplicated values
range_in_column(client, col, product)	Get range for a column
exists_row_where(client, criteria, product)	Check whether at least one row exists matching the criteria

ploomber.testing.sql.nulls_in_columns

ploomber.testing.sql.**nulls_in_columns**(*client*, *cols: str | List[str]*, *product*)

Check if any column has NULL values, returns bool

Parameters

- **client** – Database client
- **cols** – Column(s) to check
- **product** – The relation to check

Returns

True if there is at least one NULL in any of the columns

Return type

bool

ploomber.testing.sql.distinct_values_in_column

ploomber.testing.sql.**distinct_values_in_column**(*client*, *col: str*, *product*)

Get distinct values in a column

Parameters

- **client** – Database client
- **col** – Column to check
- **product** – The relation to check

Returns

Distinct values in column

Return type

set

ploomber.testing.sql.duplicates_in_column

ploomber.testing.sql.**duplicates_in_column**(*client*, *col: str | List[str]*, *product*) → bool

Check if a column (or group of columns) has duplicated values

Parameters

- **client** – Database client
- **cols** – Column(s) to check
- **product** – The relation to check

Returns

True if there are duplicates in the column(s). If passed more than one column, they are considered as a whole, not individually

Return type

bool

ploomber.testing.sql.range_in_column

ploomber.testing.sql.**range_in_column**(*client*, *col: str*, *product*)

Get range for a column

Parameters

- **client** – Database client
- **cols** – Column to check
- **product** – The relation to check

Returns

(minimum, maximum) values

Return type

tuple

ploomber.testing.sql.exists_row_where

ploomber.testing.sql.**exists_row_where**(*client*, *criteria: str*, *product*)

Check whether at least one row exists matching the criteria

Parameters

- **client** – Database client
- **criteria** – Criteria to evaluate (passed as argument to a WHERE clause)
- **product** – The relation to check

Notes

Runs a `SELECT EXISTS (SELECT * FROM {{product}} WHERE {{criteria}})` query

Returns
True if exists at least one row matching the criteria

Return type
bool

2.7 Related projects

Check out other amazing projects brought to you by the Ploomber team!

- sklearn-evaluation Plots for evaluating ML models, experiment tracking, and more!
- ploomber-engine: A toolbox for executing, testing, debugging, and profiling Jupyter notebooks
- JupySQL: Query SQL databases from jupyter with a `%sql` magic: `result = %sql SELECT * FROM table`

2.8 Community

2.8.1 Contact Us

Do you have any questions or feedback? Reach out to us:

- E-mail us at contact@ploomber.io.
- Send us a message on Slack.
- Open an issue on GitHub.

Stay up-to-date

- Follow us on Twitter.
- Subscribe to our newsletter.
- Subscribe to our YouTube channel.

2.8.2 Code of Conduct

Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation. We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

Our Standards

Examples of behavior that contributes to a positive environment for our community includes:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or e-mail address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Slack Guidelines

- Use the `#ask-anything` channel to post questions.
- When using the `#ask-anything` channel, provide sufficient detail for the community to provide an answer
- If you have a private question, you can send a private message to the community leaders (Eduardo and Ido) but bear in mind that it may take some time to get an answer.
- Do not solicit community members about your product, project, or service (even if open-source or free)
- Do not demand attention from community members by sending private messages or using `@here`, or `@channel`.

Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account or acting as an appointed representative at an online or offline event.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at conduct@ploomber.io. All complaints will be reviewed and investigated promptly and fairly. All community leaders are obligated to respect the privacy and security of the reporter of any incident.

Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, clarifying the nature of the violation and explaining why the behavior was inappropriate. A public apology may be requested.

1. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A severe violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any interaction or public communication with the community for a specified period. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any public interaction within the community.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available here.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ here. Translations are available here.

2.8.3 API Changes

We follow semantic versioning, which means **we won't introduce API incompatible changes in minor versions** (e.g., from `0.15.x` to `0.15.y`). Major versions introduce API incompatible changes (e.g., from `0.x` to `0.y`), however, Ploomber's API has been stable for over a year now, and API incompatible changes have only required minor code updates.

Deprecation policy

Whenever we introduce an API incompatible change, we add a `FutureWarning` and keep it for two minor releases before rolling out the major release.

Changelog

We keep a detailed log of changes in our CHANGELOG on GitHub.

2.8.4 Contributing

Thanks for considering contributing to Ploomber! There are many ways to do so. From fixing bugs or adding features to the core framework, improving this documentation or contributing to the examples repository.

Click on the appropriate link in the list below to learn more.

1. Core framework.
2. This documentation.
3. Examples repository.

If you have any questions, open an issue or send us a message on Slack.

2.8.5 User Statistics

The data we collect is limited to:

- The Ploomber version currently running.
- A generated UUID, randomized when the initial install takes place, no personal or any identifiable information.
- Environment variables: The OS architecture Ploomber is used in (Python version etc.)
- Information about the different product phases: installation, API calls and errors.
- For users who explicitly stated their email, we collect an email address.

How to opt-out

As an open source project, we collect anonymous usage statistics to prioritize and find product gaps. This is optional and may be turned off either by:

1. Modify the configuration file `~/.ploomber/stats/config.yaml`:

- Change `stats_enabled` to `false`.

2. Set the environment variable:

```
export PLOOMBER_STATS_ENABLED=false
```

Version updates

If there's an outdated version, ploomber will alert it through the console every second day in a non-invasive way. You can stop this checks for instance if you're running in production and you've locked versions. The check can be turned off either by:

1. Modify the configuration file `~/.ploomber/stats/config.yaml`:

- Change `version_check_enabled` to `false`.

2. Set the environment variable:

```
export PLOOMBER_VERSION_CHECK_DISABLED=false
```

2.8.6 Roadmap

These are some of the features we have in the pipeline, sorted by priority. Please help us prioritize this list; go to the related GitHub issue and comment on it.

1. Grid improvements (#522, #647)
2. Plot color changes (#650)
3. Better tracking of execution products (notebooks, models and artifacts).
4. Improve documentation of git collaboration `{{git}}` (#615).

Done

1. Better support for non-Jupyter editors like VSCode or PyCharm (available in `0.14`). *User guide.*
2. Export Ploomber pipelines to Slurm. (Note: this will be implemented in Soopervisor). (This is now in beta)
3. CLI automatic suggestions on typos (#618)
4. Scheduling via cron jobs (#422)
5. Pipeline Monitoring (#583)
6. Google Colab integration, documentation and a sample first-pipeline.
7. Managed AWS Batch deployment through github actions CI.

To send general feedback, open an issue or send us a message on Slack.

Ideas

These are some ideas we have that we haven't prioritized yet.

- Deploying your model as an endpoint.
- Artifacts reproducibility.
- Integration with Databricks.
- Integration with Great Expectations.
- Integration with Streamlit.
- Automated pipeline testing.
- Integration with data versioning tools such as LakeFS. (#414)
- Expand integration with Google Cloud (we only support uploading to Cloud Storage).
- Expand integration with AWS (we only support S3 and AWS Batch).
- Integration with Azure Machine Learning services.
- Support for Julia.

To send general feedback, open an issue or send us a message on Slack.

INDEX

E

F

G

I

K

L

N

O

P

R

S

T

U

V

W