Day 3 - Guidelines

> Today we will talk about pipelines and graphs! Our simple data loading pipeline will be the perfect use case to implement interesting data structures.

✔️ Prerequisites

- Be sure that the `day_2` directory is complete (you can copy-paste from our yesterday's suggestions if necessary)

- **Copy all the content of `day_2` directory to a new `day_3` directory: today, we will work exclusively in `day_3`**

- Update your package installation by running `pip install -e .` **from the `day_3` directory.**

📚 Resources

- To have a better understanding of the data structures we will use today, you can watch this 15-minute video, introduction basic graph concepts.

  > https://www.youtube.com/watch?v=eQA-m22wjTQ

- And you may want to read this old Python essay about graphs in Python. **Not now**, but later 😉

💡 General tips

- Every exercise has a suggested duration. Use them as an indicator as to when you should move on to the next one.
- Talk with your teammates during the workshops. Really 🙂
- Use IPython or a notebook to debug your code.

  Today, it is **not a suggestion**: you are asked to work concurrently in your IDE and in a notebook, to ease your debugging process and enjoy nice visuals.
- Use type annotation as much as you want!
- Remember: our code snippets are nothing more than suggestion.

0. Refactoring the training system

`Duration: 5-10 min`

Yesterday, we built a prediction system that lives in its own Python module, `predict`. To bring more consistency and clarity, we'll do the same with the existing training system

Create a `src/training` directory, and move the following modules inside:

- `src/data.py` → `src/training/data.py`
- `src/evaluation.py` → `src/training/evaluation.py`

- `src/features.py` → `src/training/features.py`
- `src/models.py` → `src/training/models.py`
- `src/training.py` → `src/training/training.py`

Update the `import` statements of your code base to take this change into account. **Imports must changed at least in those locations** (maybe more, depending on your implementation):

- `src/train/train.py`
- `src/run_dataset.py`
- `src/run_predict.py`
- `src/run_train.py`
- `src/prediction/predict.py`

> ⚠️ **NOTE**
>
> We are importing objects from the training system to the prediction system... It's a clear sign that our design is not perfect yet: there should be clear boundaries between them.

1. A simple sequential pipeline

`Duration: 20-30 min`

1.1 Update the `io` module

Currently, the `get_data_catalog` function has a bad separation of concerns that will make our pipeline implementation more difficult. We need 2 distincts function to load the raw CSV. We will fix it **right now!**

Remember - after yesterday's exercises, you should have access to the directories through the `context` object, located in `src/context.py`.

```
# src/io.py
...

def get_data_catalog():
    # TODO: update to use the 2 following functions
    ...
    return {'products': products, 'transactions': transactions}

def load_transactions():
    # TODO: load `transactions.csv`
    # Use the `context` object!

def load_products():
    # TODO: load `products.csv`
    # Use the `context` object!

...
```

1.2 Update the `run_dataset` module

Instead of `get_data_catalog`, the `main` function should use the new `load_transactions` and `load_products` functions.

1.3 Building your first DAG

We are now ready to implement a DAG that will take care of the data engineering part of our program.

First, create a `src/libs/dag` directory, and add a `graph.py` module inside.

> ⚠️ **IMPORTANT**
>
> We called that directory `libs` just to be clear that this is a **project within the project**. In an ideal world, you would use a 3rd-party library to implement your DAGs.
>
> We're working on it! 🙂

The `Graph` class will be very simple:

- its only attribute (state) is `edges`, a mapping acting as our Adjacency List. We use a `defaultdict` from the standard library. The keys of the dictionary will be every nodes currently in the graph, and the corresponding values will be a list of all the adjacent nodes.
- its only method (behaviour) is `add_edge`: the ability to update its own state by adding edges. This relation is directed (it's a DAG...) from the target node toward the source node, which reads *target node is dependent on source node*.

> 🗒️ **NOTE**
>
> The graph's nodes are the executable functions. The edges represent the dependencies between those functions.
>
> For example, with our current implementation, `get_daily_transactions` **depends on** `load_transaction`. Hence, our modelization will put an edge between those two functions.

```
# src/libs/dag/graph.py
# You may find `defaultdict` from the standard libary useful:
# If so, just import it :)
# https://docs.python.org/3/library/collections.html#collections.defaultdict

class Graph:
```

```
    def __init__(self):
  # TODO: `edges` should be a mapping.
        # The mapping keys must be the nodes, and
        # their corresponding values must be
        # a list of all nodes adjancent to the given key.
        self.edges = ?

    def add_edge(self, source, target):
        # TODO: add the source node to the list of target's
        # adjacent nodes.
```

1.4 Refactor `data.py`

To better represent what your functions will do in a moment, rename them:

- `_get_daily_transactions` → `_compute_daily_transactions`
- `_get_weekly_transactions` → `_compute_weekly_transactions`

1.5 Update the `dataset` pipeline

The `build_dataset` function must now use our graph, and build edges between its nodes.

First, rename the functions `_get_daily_transactions` and `_get_weekly_transactions` to `_compute_daily_transactions` and `_compute_weekly_transactions`: those names are better suited to what they are actually doing.

```
# src/training/data.py
...
from src.io import load_products, load_transactions
from src.libs.dag.graph import Graph
...

def build_dataset():
    # TODO: remove all the existing logic, except logging
    pipeline = Graph()
    # TODO:
    # 1. add an edge to the graph, between `load_transaction`
```

```
    #    and `_compute_daily_transactions`
    # 2. add an edge to the graph, between `_compute_daily_transactions`
    #    and `_compute_weekly_transactions`

    # TRAINING NOTE: That's it for now!
    # We go step by step, and we'll update it later.

    return pipeline
```

NOTE

As you may have guessed, your `dataset` pipeline is now broken! That's expected. We want to make progress step-by-step.

Before implementing the whole logic (*eg* merging *products* and *weekly transactions)*, we want to make sure that our graph is working as expected.

2. Visualizing the pipeline

`Duration: 20 min`

2.1 A first draft

One of the main feature of graphs it their inherent ability to be represented visually.

It's time to work interactively in a Jupyter notebook environment, a very convenient tool for debugging.

In notebook code snippets, a notebook cell will be represented enclosed in `##########`

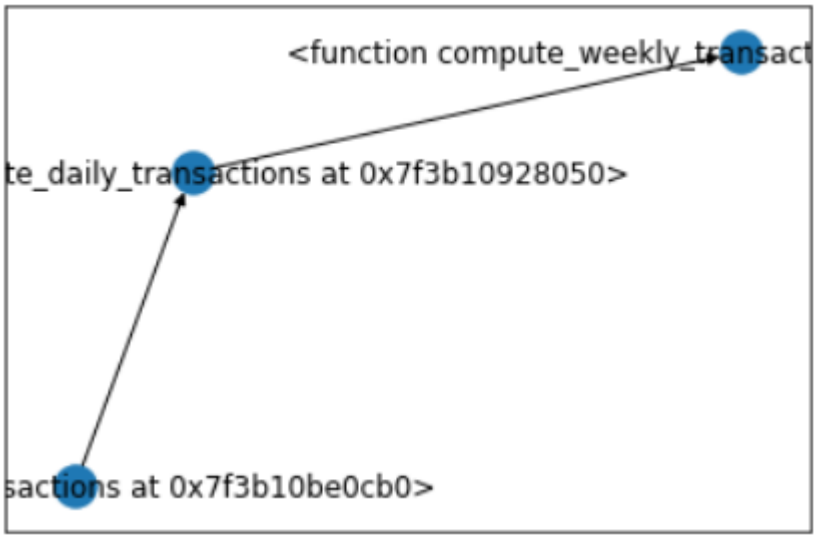
> ⚠️ **WARNING**
>
> Be careful to run the Jupyter Notebook process from the same virtual environment as your project! Otherwise, importing from `src` to the notebook will be impossible.

We'll use NetworkX, an open-source Python library built to work with graphs data structures.

```
# in a Jupyter notebook
######################################################################
%load_ext autoreload
%autoreload 2
######################################################################
%matplotlib inline
######################################################################

import matplotlib.pyplot as plt
import networkx as nx   # install it if missing: !pip install networkx

######################################################################
# Our pipelines need an up-to-date context to work.
from src.config.config import get_config
from src.context import context


context.environment = 'development'  # or 'dev', depends on your config files
context.config = get_config(context)
######################################################################

def draw_graph(graph, *, ax=None, save_path=None):
    if ax is None:
        ax = plt.axes()
    # NOTE:
    # We need `.reverse` here because we have reversed
    # our adjacency list for convenience. You shouldn't
    # care about that detail.
    nx_graph = nx.DiGraph(graph.edges).reverse()
    nx.draw_networkx(nx_graph, ax=ax)
    if save_path is not None:
        ax.figure.savefig(save_path)

    return ax

######################################################################

from src.training.data import build_dataset  # the graph used for testing
```

```
graph = build_dataset()

#########################################################################
draw_graph(graph);  # use the ';' for nicer display
```

The result must be something similar to this:



This is pretty cool, but could be much better... For instance, instead of the Python `repr` of the functions, it would be better to have their names.

2.2 Visualizing graph with function names

To visualize the function names instead of their Python representations, we need to build a mapping that will associate function names with their memory. Here is a quick summary of the situation:

```
# The current graph structure used for visualisation is this:
adjacency = {
    func_A: [func_B, func_C],
    func_B: [func_C]
}

# and we'd prefer this:
```

```
adjacency = {
    'name_of_func_A': ['name_of_func_B', 'name_of_func_C'],
    'name_of_func_B': ['name_of_func_C']
}
```

We need to implement a helper function to build this new graph representation. Just copy/paste the following snippet in a new file called `src/libs/dag/utils.py`

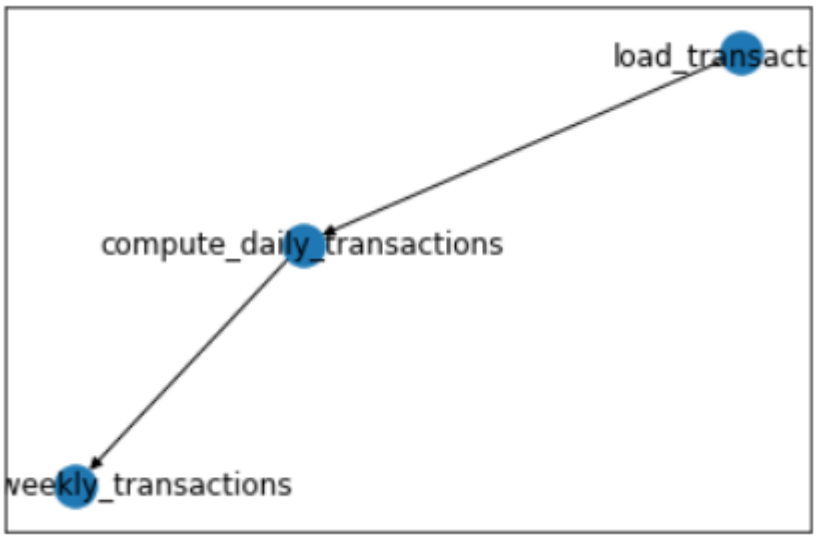
```
# src/libs/dag/utils.py
from typing import Callable, Mapping, Dict, Iterable, Any

def map_adjacency(
    func: Callable,
    adjacency: Mapping[Any, Iterable])
 -> Dict[Any, Iterable]:
    """An improved version of the built-in map.

    Return a dictionary whose keys are `func(key)` and
    values are `func` applied to every element of
    the given mapping values.

    Args:
        func: a callable
        adjacency: a mapping that has exclusively iterable values.
    """
    return {
        func(key): [func(v) for v in values]
        for key, values in adjacency.items()
    }
```

Then, back in your notebook, rewrite the `draw_graph` function:

```
# in your existing notebook
from src.libs.dag.utils import map_adjacency

################################################################

def draw_graph(graph, *, ax=None, save_path=None):
    if ax is None:
        ax = plt.axes()

    # We use our utils function here: using the __name__
    # special attributes that is attached to every Python
```

```
    # functions or classes.
    adjacency = map_adjacency(lambda f: f.__name__, graph.edges)
    nx_graph = nx.DiGraph(adjacency).reverse()
    nx.draw_networkx(nx_graph, ax=ax)
    if save_path is not None:
        ax.figure.savefig(save_path)
    return ax

################################################################
draw_graph(graph);
```

And now we have the expected result!



2.3 Using `pyvis` - bonus

TIMER WARNING

This is a bonus step. Skip it unless you're comfortable with the clock.

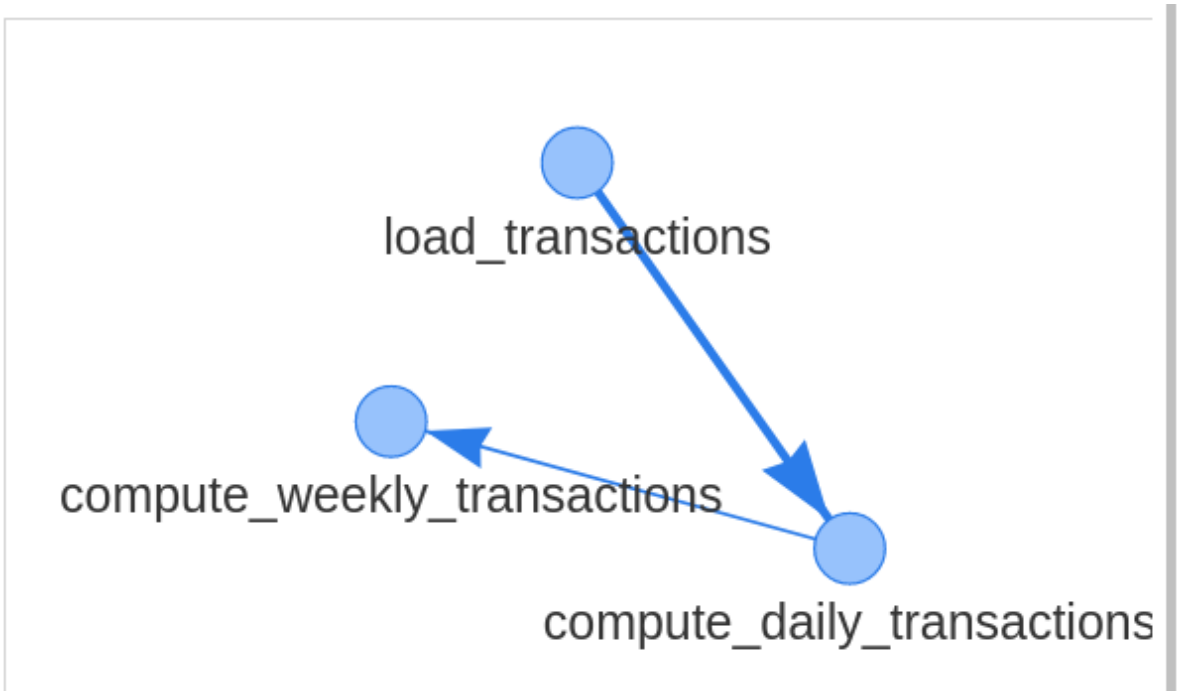
Using `pyvis` can make your visualisation experience cooler 😎.

First, install `pyvis` on your virtual Python environment: `pip install pyvis`

Then, update your `draw_graph` function so it can be used both with `pyvis` and `Matplotlib`.

```
# in your existing notebook
from pyvis.network import Network

#############################################################################

def draw_graph(graph, *, path=None, backend='pyvis', ax=None):
    """
    Draw a graph using prefered backend.

    Args:
    -----
        graph: a `dag.graph.Graph` object
        path (Path|str): where to save the file (required for pyvis backend)
        backend: 'pyvis' or 'matplotlib'
        ax (matplotlib.pyplot.axes): optional for matplotlib
    """
    adjacency = map_adjacency(lambda f: f.__name__, graph.edges)
    nx_graph = nx.DiGraph(adjacency).reverse()

    if backend == 'matplotlib':
        if ax is None:
            ax = plt.axes()
        nx.draw_networkx(nx_graph, ax=ax)
        if path is not None:
            ax.figure.savefig(path)

        return ax

    elif backend == 'pyvis':
        assert path is not None, (
            "'path' is required if you're using pyvis backend."
        )
        net = Network(directed=True, notebook=True)
        net.from_nx(nx_graph)

        return net.show(str(path))

    else:
        raise NotImplementedError("Unknown backend.")


#############################################################################

draw_graph(graph, path='graph.html')  # use current notebook directory
```

And now you have an even nicer interactive visualization! (it's not interactive in here... don't ask too much!)



> Note the directions of arrows... It gives us clear hints about which function depends on the others.

3. A better Graph object

`Duration: 10 min`

Before being able to execute our graph, we need to be able to:

- Get an iterable of a node's dependencies
- Sort the graph in topological order (*ie*, which nodes must be executed before the others)

3.1 Helper function

Convert our custom graph to a NetworkX graph:

```
# src/libs/dag/utils.py
import networkx as nx
...

def convert_to_networkx(graph):
    # need to reverse because of our implementation choices
    return nx.DiGraph(graph.edges).reverse()
```

As you can see, we are now using `networkx` in the project itself, not only in the debugging notebook... It's time to add it to the requirements:

```
# requirements.txt
...
networkx==2.5
```

3.2 Enriching the `Graph` class

Add 2 methods to your `Graph` objects, useful for the behaviour we are currently missing:

- `get_node_dependencies`
- `topo_sorted`

```
# src/libs/dag/graph.py
import networkx as nx

from src.libs.dag.utils import convert_to_networkx

class Graph:

    def __init__(self):
        self.edges = # you did that in a previous exercices
```

```
    def add_edge(self, source, target):
# and you also did that one too :)

    def get_node_dependencies(self, node):
        # TODO: return the list of all the dependencies
        # of the given node, or an empty list if it has none.
        return ?

    def topo_sorted(self):
        nx_graph = convert_to_networkx(self)
        return nx.algorithms.dag.topological_sort(nx_graph)
```

4. Graph execution

`Duration: 30-40 min`

Visualizing the graph is cool, but pretty useless, right? In this exercise, we'll build a simple executor implementation.

In this exercise, you will work both in the notebook and in your project's code files. Do not hesitate to explore, execute, make changes, etc...

4.1 A simple function

In your notebook, create a dummy function `execute_graph` that we will use to build our executor incrementally.

First, let's iterate over each node in topological order, and print it:

NOTE

Our logging configuration is not setup to work in this notebook... That's OK. We'll just use plain `print` statements when we need it. We don't want you to miss the point!

```
# in your notebook

def execute_graph(graph):
    for node in graph.topo_sorted():
        print(node)  # useless, but enough for now.
```

And you should see your nodes printed out in the console, **in this order:**

1. `load_transactions` - the "root" function
2. `compute_daily_transactions`
3. `compute_weekly_transactions`

4.2 Some kind of execution

To correctly manage the graph execution, we need to manage:

- **State -** the intermediate computations
- **Behaviour -** the sequential execution of the nodes

Hence, the best tool to solve this problem is probably another class! Encapsulate your function in an `Executor` class:

```
# src/libs/dag/executor.py
import logging

logger = logging.getLogger(__name__)

class Executor:

    def execute(self, graph):
        for node in graph.topo_sorted():
            logger.debug(node)  # temporary, of course!
```

Now that we have an "executor" (we can't seriously call that an *executor* at the moment, but you get the point 😜), we can instantiate it where it's needed, then run

an execution:

```
# src/run_dataset.py
...
from src.libs.dag.executor import Executor
...

def main():
    ...
    pipeline = build_dataset()
    executor = Executor()
    executor.execute(pipeline)
    ...
```

Run the `dataset` pipeline with your CLI to see if you get the expected log. Your console should look similar to this (watch the order!):

```
$ python -m src dataset
[TIMESTAMP]      INFO    Building dataset...
[TIMESTAMP]      DEBUG   <function load_transactions at 0x7f5650c7f820>
[TIMESTAMP]      DEBUG   <function _compute_daily_transactions at 0x7f5650c7fd30>
[TIMESTAMP]      DEBUG   <function _compute_weekly_transactions at 0x7f56377f9ca0>
```

4.3 A real first execution

OK, it seems that our executor is working... except that it doesn't execute anything!

Remember that we chose a model where the nodes of the graph are simply callable functions. **To execute a node, you just need to call it**.

Update your `Executor` class accordingly.

```
# src/**/executor.py

class Executor:
...
    def execute(self, graph):
        for node in graph.topo_sorted():
      # TODO:
            # 1. execute the node
            # 2. store the result in an `output` variable
            # 3. print the output as a debug log message.
```

Now, run the `dataset` pipeline from the CLI, and watch the logs carefully... ⛈️

> ⛔ **NO TRESPASSING**
>
> Before proceeding to the next part, you **have to get your trainer's green light**.
>
> Send me a message on Slack, or ask me to join you, and explain to me what happens, and **why is your pipeline crashing.**

🚧 🚦 ❓ ⬆️

4.4 Handle executor's state

The executor needs to keep some records of its inner state so it can pass the last computation output to the next one.

After an execution, it is important that **we can map the output of a given node to the node itself.** Reminder: in our case, a node is nothing more than a function.

```
# src/**/executor.py

class Executor:

    def __init__(self):
        # TODO:
        self._state = # ?

    def execute(self, graph):
        for node in graph.topo_sorted():
            # TODO:
            # 1. Keep the node's output in the `_state` attribute
            # 2. Log the output and the node, for debugging
```

Run the `dataset` pipeline via the CLI. **It will still crash,** but you should see at least one output before it raises an error.

> We're not there yet, but we are getting closer! ⏳

4.5 Improve state management

Because we're executing the graph in topologically sorted ordering, we know that a node's **dependencies have been computed before the execution of the node** itself, and that their outputs were saved somewhere in the executor's state (cf. 4.4)

Before each node execution, we need to:

- Fetch all of the node's upstream dependencies
- Get these dependencies outputs, so we can use them as node's inputs

```
# src/**/executor.py

class Executor:

    def __init__(self):
        self._state = {}

    def execute(self, graph):
        for node in graph.topo_sorted():
            # TODO: store all the node's dependency in `deps`
            deps = ?
            upstream_data = (self.__state[d] for d in deps)
            # TODO: Using data from upstream nodes,
            # update the instance's state by storing the output
            # of the node in the dictionary
            self._state[node] = ?
```

Wait a minute before executing the `dataset` pipeline again...

4.6 Capturing last node's output

NOTE

Is it the best idea? Maybe not. For instance, it won't work for graphs with multiple terminal nodes... But to keep it as simple as possible, it's enough for our use case.

What we need now is to store the **last** computed node, so we can easily fetch its output in the executor's state. Once we get its input, we can simply return it.

```
# src/**/executor.py

class Executor:

    def __init__(self):
        self._state = {}
```

```
    def execute(self, graph):
        logger.info(f"Running execution of {graph}...")
        last_computed_node = None  # defensive, in case graph is empty.
        for node in graph.topo_sorted():
            deps = # You did it in 4.5
            upstream_data = (self.__state[d] for d in deps)
            self._state[node] = # You did it in 4.5

            # Note: 'last_computed_node' is an invariant.
            # Cf. https://stackoverflow.com/a/112088
            last_computed_node = node

        logger.info(f"Successfully executed {graph}.")
        return self._state.get(last_computed_node)
```

> Now is a good time to run the `dataset` pipeline... This time, you should be happy with the result: it should execute without any error! 😃

5. About branching

`Duration: 15-20 min`

Lot of work for a small result... Indeed, we already knew how to execute functions sequentially.

For simple data flows, a DAG like the one we just built is overkill. Basic imperative Python or functional composition will work perfectly to handle these use cases.

> DAGs become an interesting tool when data flows get more complex.

In a real-life ML project, each node can have multiple parents (dependencies) or children. In our simple use case, the `merge_transactions_with_product` node has indeed

2 dependencies: it relies on both

- `load_products` and
- `_compute_weekly_transactions`

to return its output. We can start leveraging the power of our `Graph` implementation.

5.1 Finishing up `build_dataset`

If you remember, we left the `build_dataset` refactoring unfinished, as we wanted to try out our graph features to ensure we were not working for nothing... **Now that we know our graph is working,** it's time to complete this function.

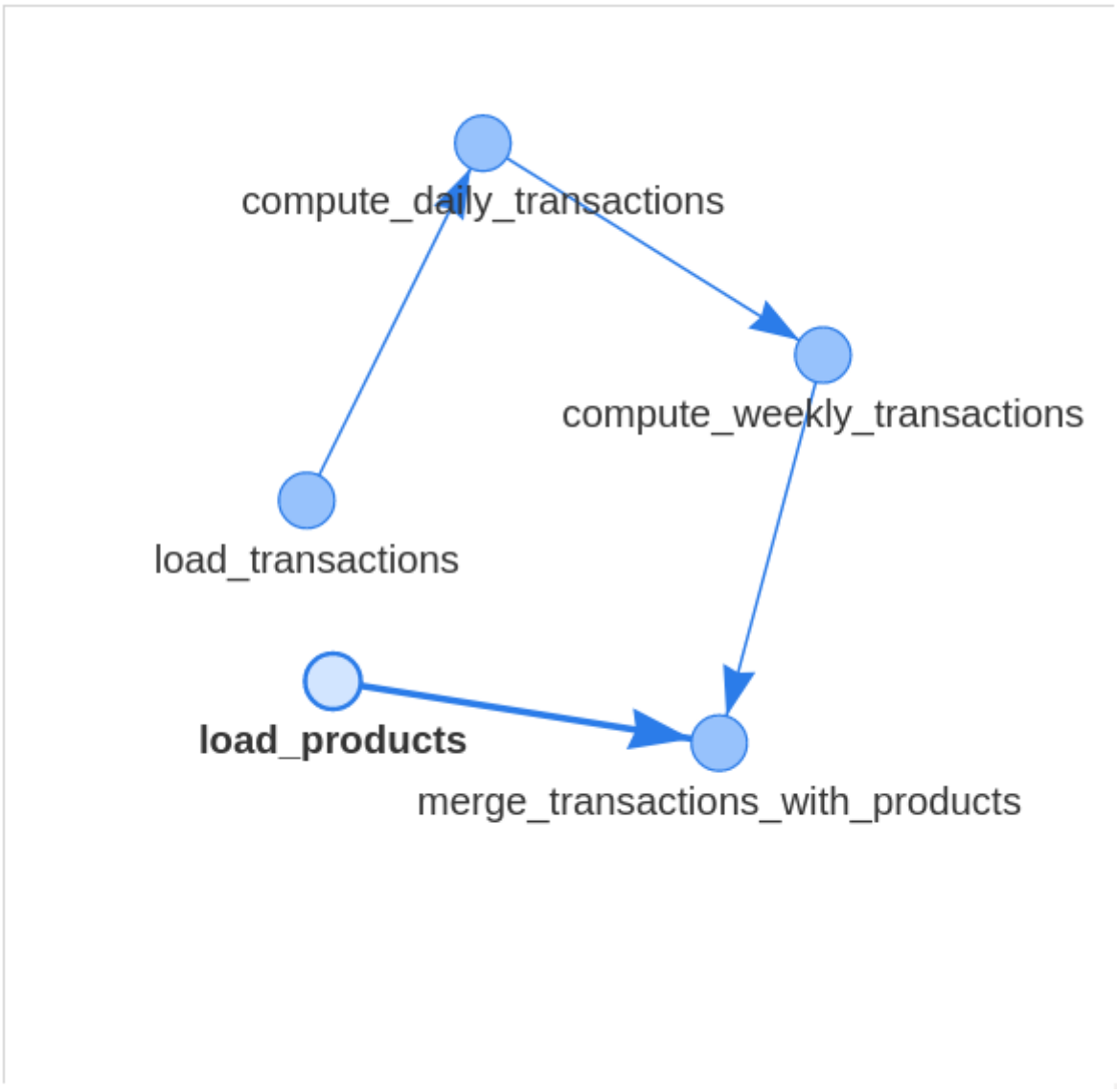
```
# src/training/data.py
...

def build_dataset():
    pipeline = Graph()
    pipeline.add_edges(
        load_transactions,
        _compute_daily_transactions
    )
    pipeline.add_edges(
        _compute_daily_transactions,
        _compute_weekly_transactions
    )
    # TODO: complete the graph with the missing parts required
    # to build the complete dataset. Hint: there are 2 missing edges.

    return pipeline
...
```

5.2 Debugging in the notebook

It's time to visualize your complete "data engineering" pipeline. Get back to the notebook, and run a visualization. (if you want a fancier visualization than the one you have, get back to 2.3, and implement the `pyvis` visualizer).

```
# in a notebook
from src.training.data import build_dataset

graph = build_dataset()

# `draw_graph` was implemented in exercise 2
draw_graph(graph, path='graph.html')  # pyvis
# or
draw_graph(graph, backend='matplotlib');  # matplotlib
```

You should see something like see: note the branching!



Now that you have verified that your graph looks exactly like what is expected, you can execute it to see its final output:

```
# in your notebook
from src.libs.dag.executor import Executor

executor = Executor()
executor.execute(graph)  # graph = build_dataset()
```

Enjoy this beautiful output dataframe 🌻

5.3 Final step

It's time to put the cherry on top of the cake.

Create a new task that will deal with saving the dataset on disk. In `io.py` we already have the lower-level function (the one that takes care of the actual writing on disk). Our task here will just wrap this existing function so it can be fed directly from the graph executor.

```
# src/training/data.py
from src.context import context
from src.constants import DATASET


def build_dataset():
    ...
    # TODO: add the final edge to your graph

    return pipeline


def _save_dataset(dataset):
    path = context.dirs.intermediate_data / DATASET
    return save_dataset(dataset, path=path)
```

NOTE

You can relaunch an interactive vizualisation if you want to see your full working graph.

Now, make sure you **delete any file located in** `data/intermediate/` and run the `dataset` pipeline from the CLI.

```
$ python -m src dataset
```

Now check in `data/intermediate`: did you successfully generate the dataset? If so, congratulations 🥳

BONUS

6. Making our custom objects more *Pythonic*

`Duration: 30 min`

As you can imagine, we barely scratched the surface of the power of DAGs... But let's step back a little.

We defined interesting custom data structures, but based on their essence, we could certainly make them more Pythonic: Python users of our classes that don't know the implementation details should be able to use them intuitively.

For instance, here is a list of what would make sense for our `Graph` objects:

1. First, we should give a name to our graphs, so we can easily represent them when debugging, or in logs.
2. A `Graph` could be an iterable: iterating over it would mean getting all the nodes sequentially, in topological order.
3. We should be able to use a `Graph` in sets, or as dictionary keys
4. Comparing 2 graphs makes also sense: 2 graphs must be considered as equal if they both have the same nodes, and the same relations between those nodes

```
# src/**/graph.py

class Graph:

    # TODO:
    # Implement special methods to make our
    # graph objects pythonic. All 'Graph' instances
    # must comply to the following interface, given below
    # as an example.

# ----------------------------
# Example, from a Python shell
# ----------------------------
>>> graph = Graph("gamma graph")
>>> graph
Graph("gamma graph")
>>> graph.add_edge(function_1, function_2)
>>> graph.add_edge(function_2, function_3)

>>> other_graph = Graph("another graph")
>>> other_graph.add_edge(function_1, function_2)
>>> other_graph.add_edge(function_2, function_3)

>>> graph == other_graph
True  # they share the same nodes and edges
>>> graph.add_edge(function_3, function_4)
>>> graph != other_graph
True  # they don't share the same nodes anymore

>>> for node in graph:
        print(node.__name__)  # assuming that all nodes are functions

>>> descriptions = {
    graph: "This graph is used to do this, and that.",
    other_graph: "Another very useful graph."
}
>>> descriptions[graph]
'This graph is used to do this, and that.'
```

```
>>> graph_3 = Graph()
>>> if not graph_3:
        print("This graph is empty!")
This graph is empty!

>>>
```

Congratulations!