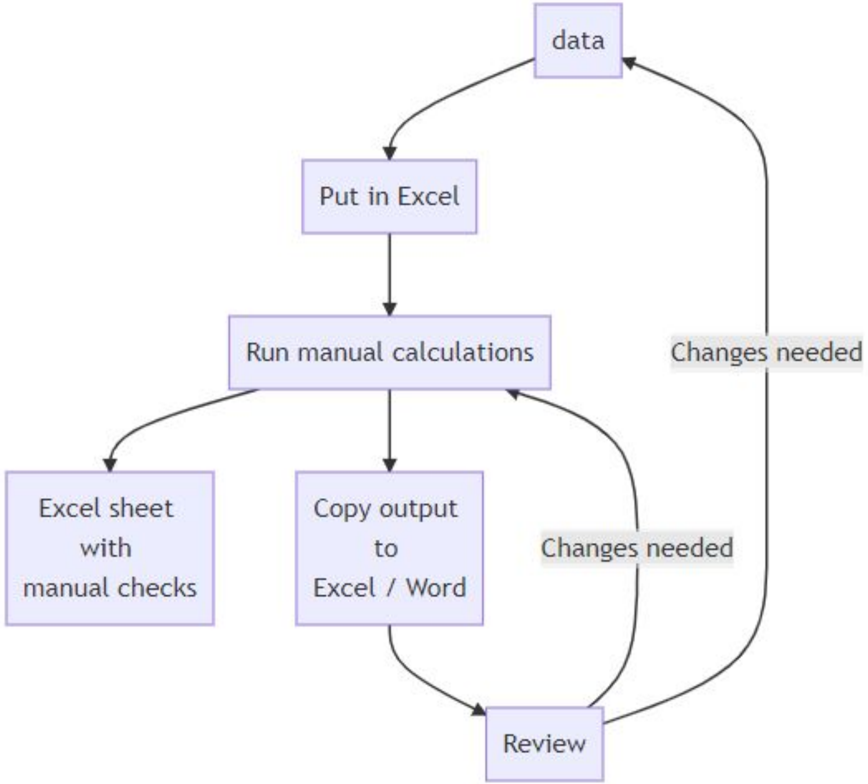
RAPping in the public sector

Shrividya Ravi
Ministry of Transport

Part 1: simple RAP

The ubiquity of bad processes



RAP as a paradigm

Reproducible Analytical Pipelines (RAPs) are automated statistical and analytical processes. They incorporate elements of software engineering best practice to ensure that the pipelines are reproducible, auditable, efficient, and high quality.

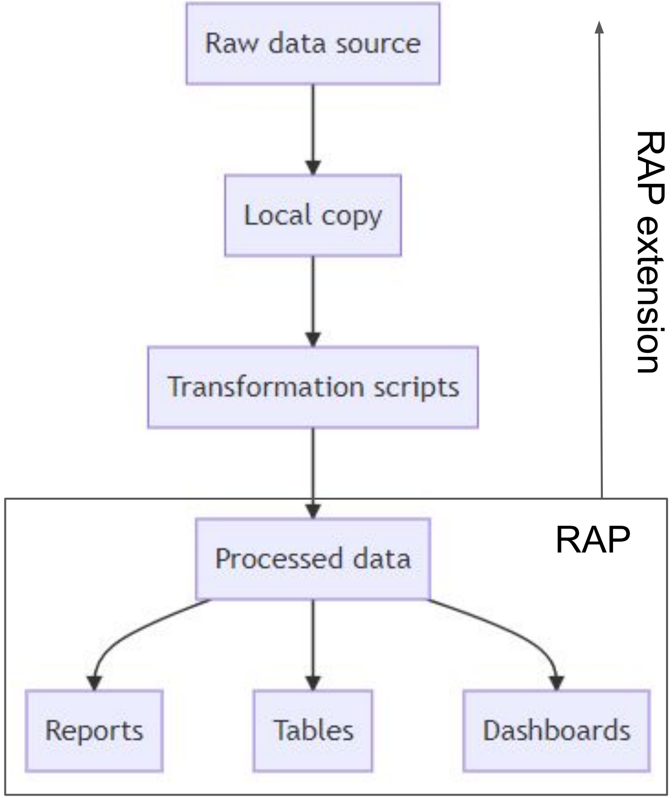
Practices include:

- substituting manual steps with code
- using modern, open source programming languages
- converting raw data to statistical output into pipelines / workflows
- using version control to keep records of development
- bringing in code review practices

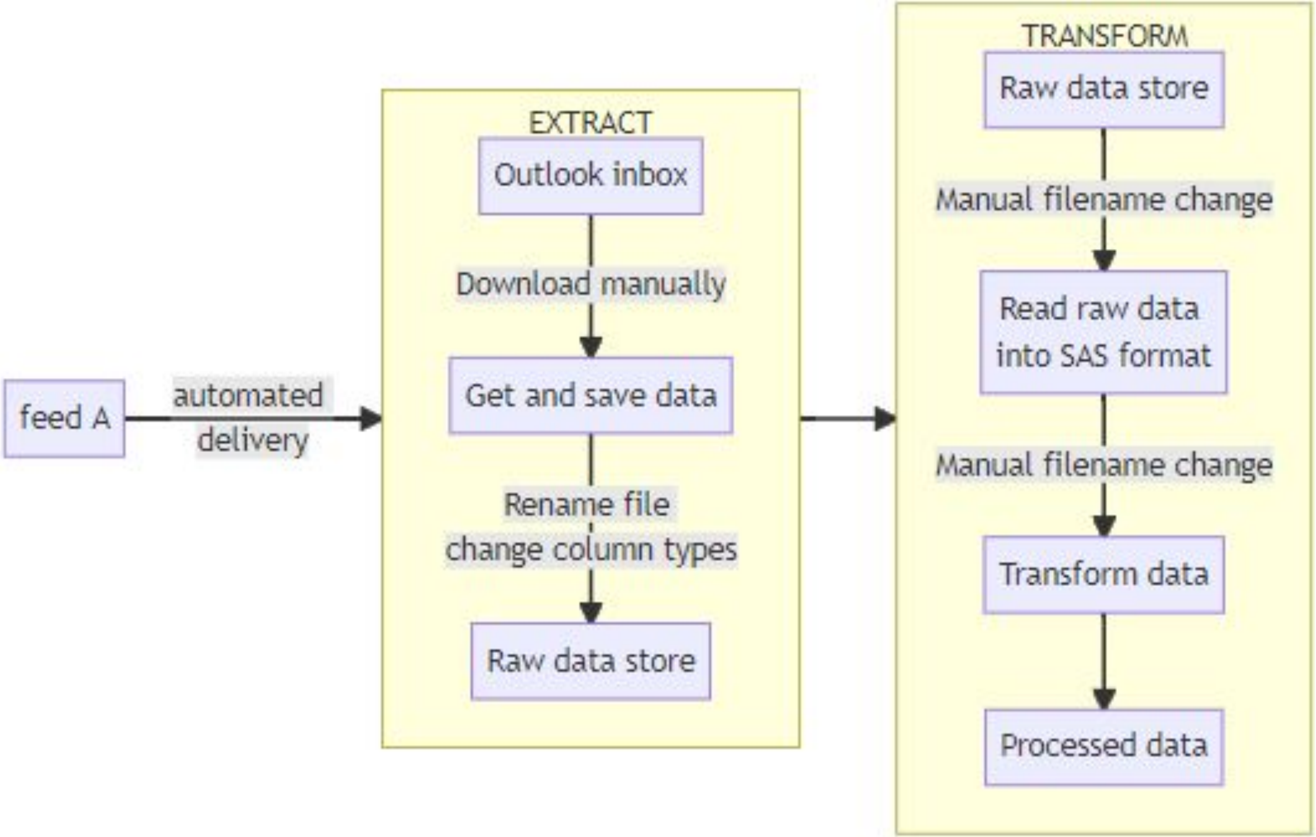
[Reproducible Analytical Pipelines – GSS (civilservice.gov.uk)]()

RAP to include data engineering

- RAP focuses on converting data from a commonly-managed data store into analytical outputs.
- In infrastructure-poor environments:
  - Data is often not in an accessible place
  - No automated process to transform it in a form fit for subsequent RAPping.
- **Concepts of RAP** need to be brought into the **data engineering domain**.



Sample legacy code



Sample legacy code

- Complexities in the code
  - Non-trivial to understand processing (e.g. unfamiliar paradigm)
  - Non-trivial business rules and algorithms
- Challenging data processing steps
  - Getting raw data from emails
  - Rolling window processing (or lack thereof)
  - Managing latent data

RAPping with legacy code?

- Two approaches:
  - Refactor completely to the RAP paradigm
  - Create interim pipelines that don’t much change the code
- Interim pipelines:
  - Allow processing work to be done much faster
  - Reduce the mental overhead of managing manual processes
  - Give more breathing room for developing better solutions

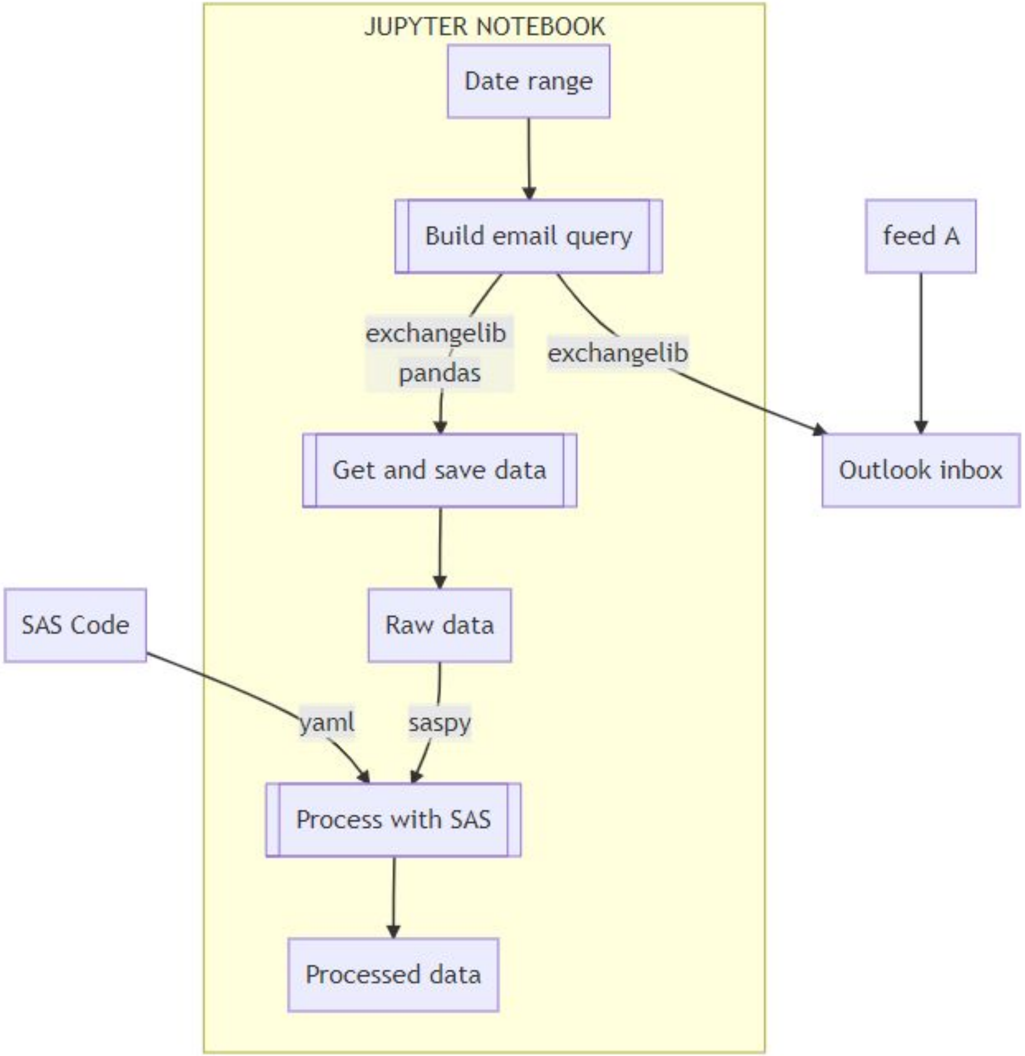
Pipelining legacy code with snake oil

Why python?

- Python is a modern, multi-paradigm, evolving, open source programming language.
- Widely used across many domains - from web development to data science.
- Due to its breadth of use and popularity, there is an incredible ecosystem of packages.

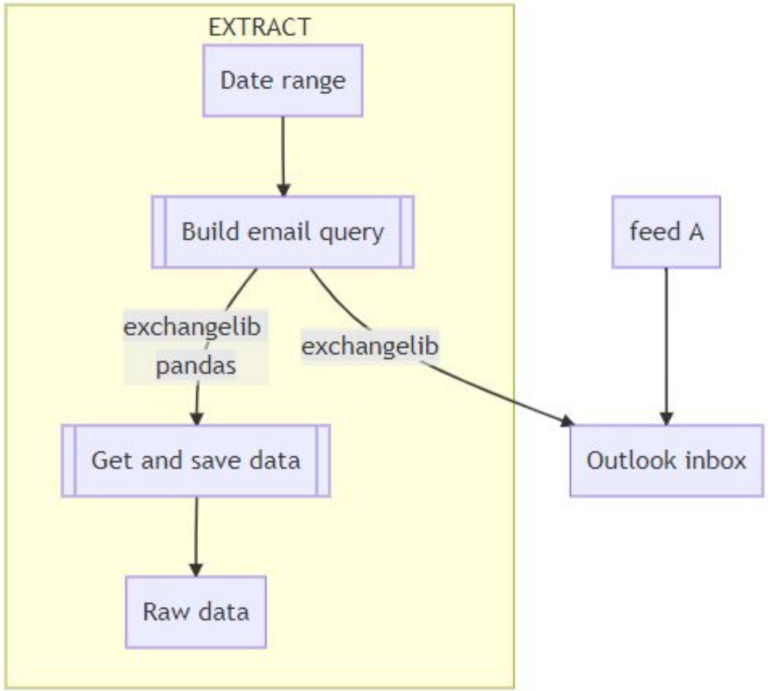
Python as glue

- Instead of manual adjustments steps, Python allows for programmatic “gluing”
- Some of the gluing is facilitated by packages, others just by basic Python functionality



Getting data with APIs - exchangelib

- Application programming interfaces (APIs) allows applications to communicate with each other.
- The Outlook email program has a rich API behind it called Exchange Web Services (EWS).
- Applications (like our RAP extract data application) can send EWS queries to push or pull data to Outlook objects like emails, contacts and calendars.



[exchangelib | Python client for Microsoft Exchange Web Services (EWS) (ecederstrand.github.io)]
[pandas - Python Data Analysis Library (pydata.org)]

Code example - exchangelib

```
# Query to get weekly automated emails from POAL
# start, end are pre-defined date ranges
weekly_poal_query = figs_account.inbox.filter(
    datetime_received__range=(start, end),
    sender="BI-DWSupport@poal.co.nz",
    has_attachments=True)

# Go through all results from email query
for i, email in enumerate(weekly_poal_query):
    # loop through all attachment
    for attachment in email.attachments:

        # if attachment, then save file
        if isinstance(attachment, ex.FileAttachment):

            # Filename and data path
            filename = "weekly_poal_data" + date.today() + ".zip"
            local_path = os.path.join(data_path, "zip_files", filename)

            # Save attachment
            with open(local_path, 'wb') as f:
                f.write(attachment.content)
```

Running SAS through Python

There are two ways or running SAS outside the SAS program:

- With a SAS kernel in Jupyter
- Through SASPy

SASPy is officially supported by SAS and, available as an open source package. The library seems to be well-maintained and well-documented.

SASPy — saspy 3.7.2 documentation (sassoftware.github.io)

Running SAS through Python

At its core, SASPy is capable of **creating a SAS session and sending code to it for execution**, as well as returning the output (logs and other output) to the controlling Python script. Yet it is also a **powerful generator of SAS code**, which means that it **offers methods, objects, and syntax for use directly in idiomatic Python that it can then automatically convert to the appropriate SAS language** statements for execution. In most cases, SAS procedures or steps are mapped directly to Python methods as a one-to-one equivalent.

Value of running SAS through Python

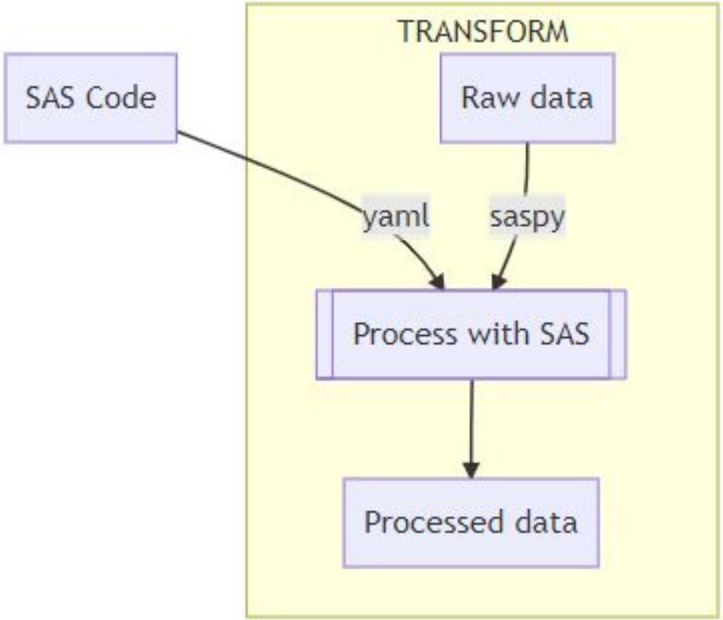
Incremental refactoring (pain management) principle

- When the (SAS) codebase is complex, large or both, it's convenient to instead just make it run easier.
- This approach allows to incremental refactoring - allowing hard-to-convert code to remain in SAS while moving easier code to Python.

Breaking up SAS code into chunks

To run existing SAS scripts, via Python, where parts need manual amendments needs an extra step. The easiest solution so far has three main steps:

- Breaking up a SAS script into yaml chunks for "immutable" components
- "Mutable" components are created in python
- The immutable and mutable are brought together with Python's f-strings



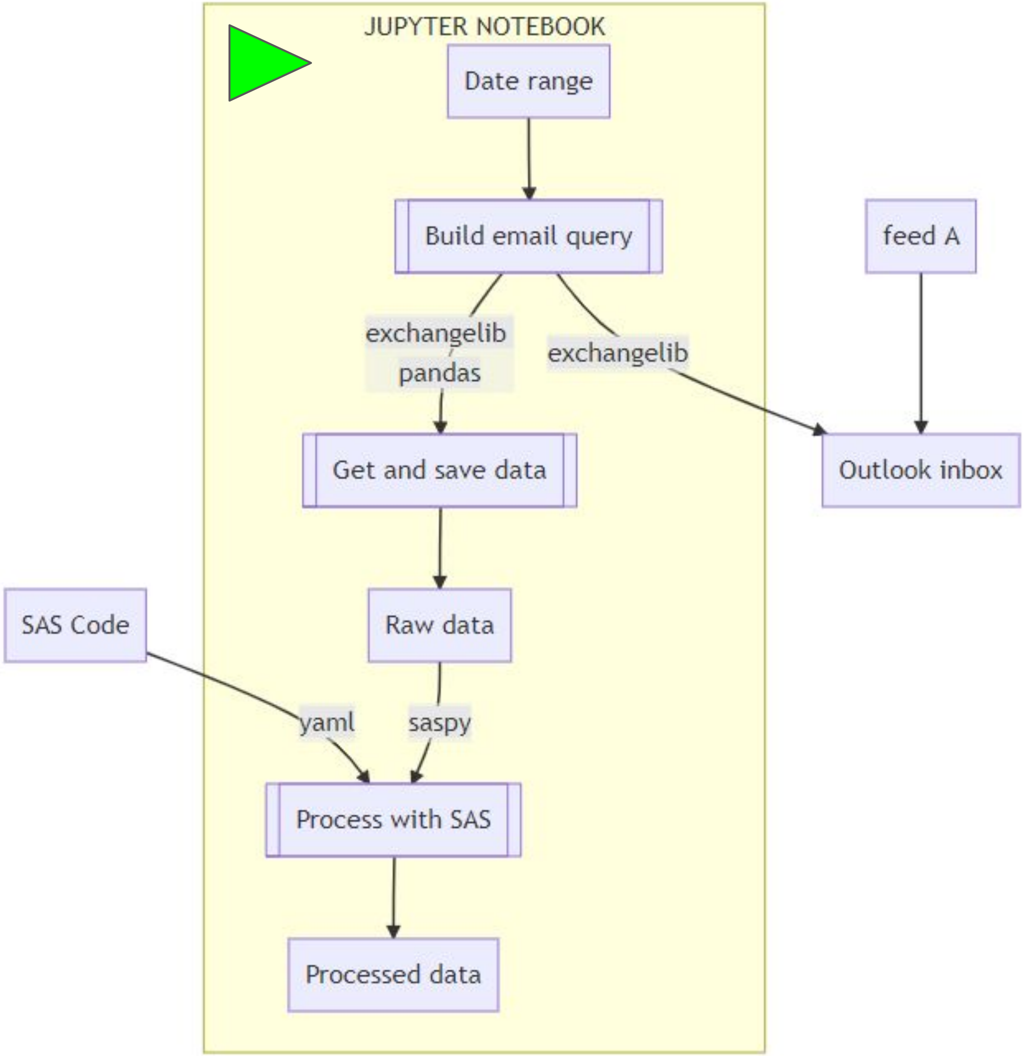
[yaml/pyyaml: Canonical source repository for PyYAML (github.com)](#)
[Python 3's f-Strings: An Improved String Formatting Syntax (Guide) – Real Python](#)

Breaking up SAS code into chunks - code

```python
# open list of new data filenames after downloading from Outlook
files_list = open('..\sas_code\combine\files_list.sas').read()

# open data processing code chunks
with open(r'..\sas_code\combine\poal_combine_sas.yaml') as file:
    sas_code = yaml.full_load(file)

# create code blocks by function
preamble = sas_code['preamble']
coarri_combine = sas_code['coarri_combine']
codeco_combine = sas_code['codeco_combine']

# create sas query - concat code blocs
sas_query = f"""{preamble}{files_list}{codeco_combine}"""
res = sas.submit(sas_query)
```

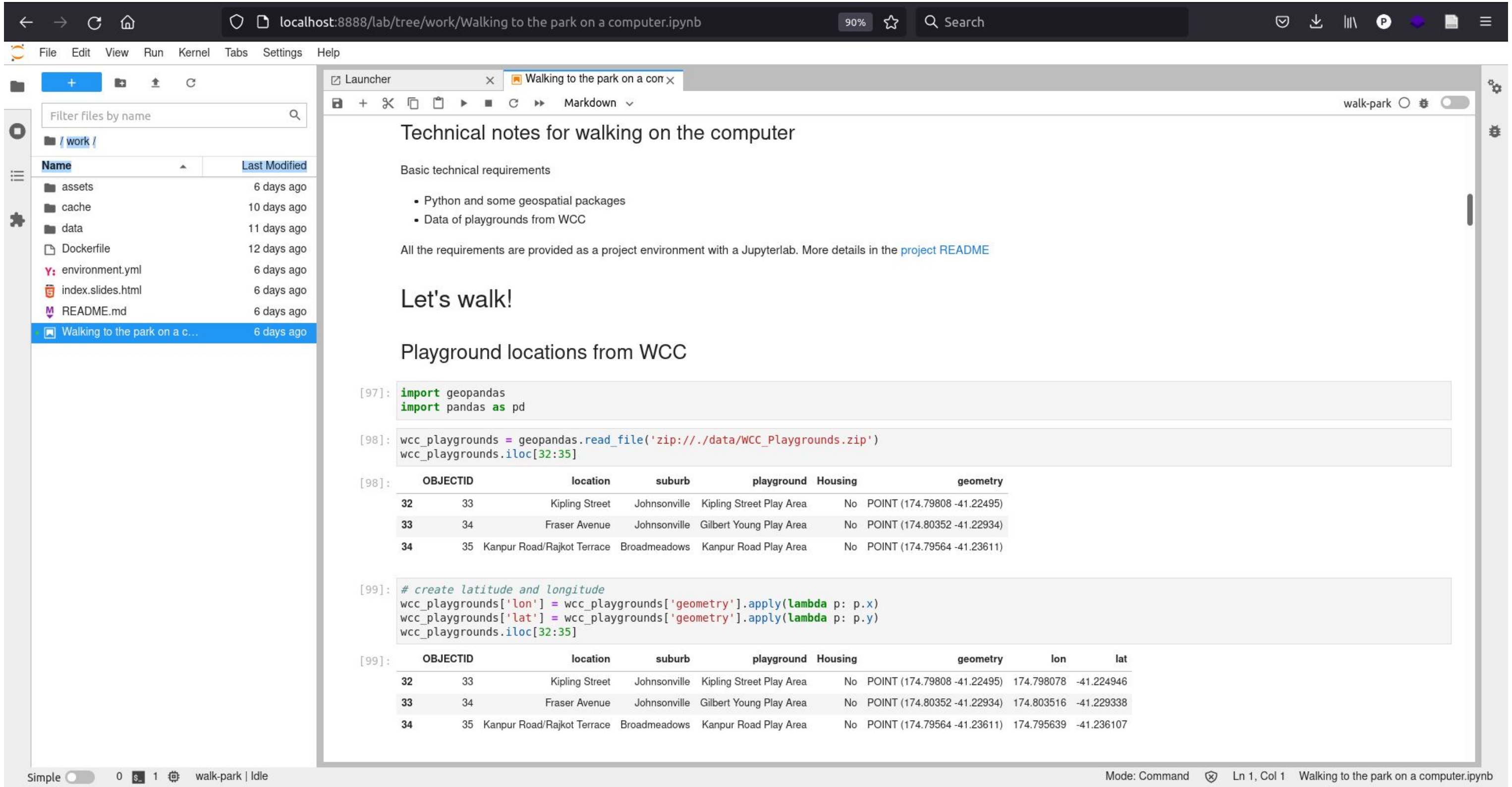
One-click linear pipelines

- Put code of pipeline steps into cells of a Jupyter notebook
- Run all steps with “one click”
- Jupyter notebooks can print outputs, checks, visualisations in a clean IDE interface



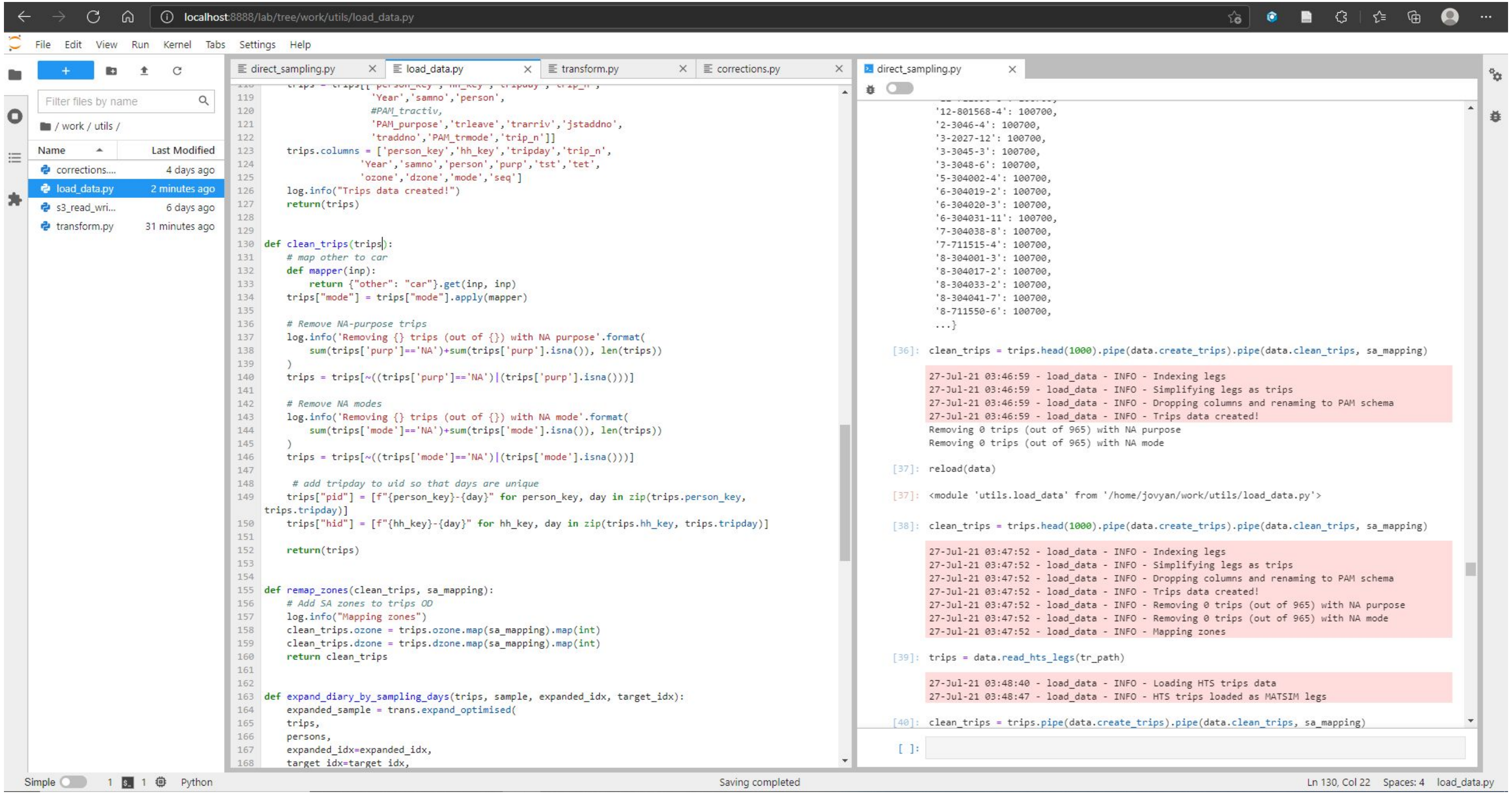
[Project Jupyter | Home]

Jupyterlab IDE and Jupyter notebooks



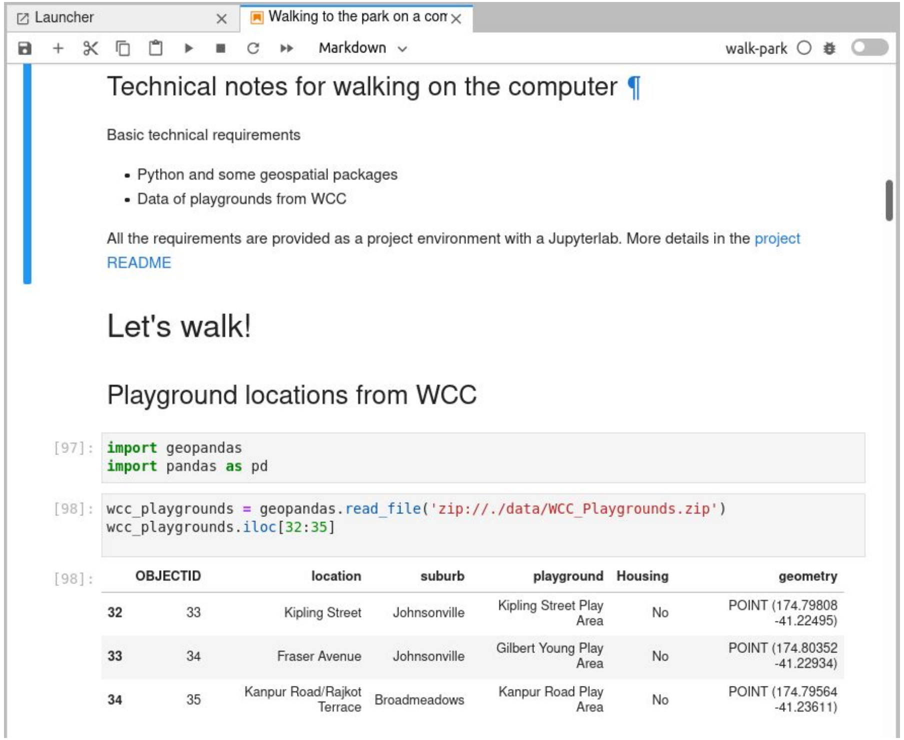
[JupyterLab Documentation — JupyterLab 3.1.0 documentation]()

Jupyterlab IDE



Jupyter to bind, execute and document

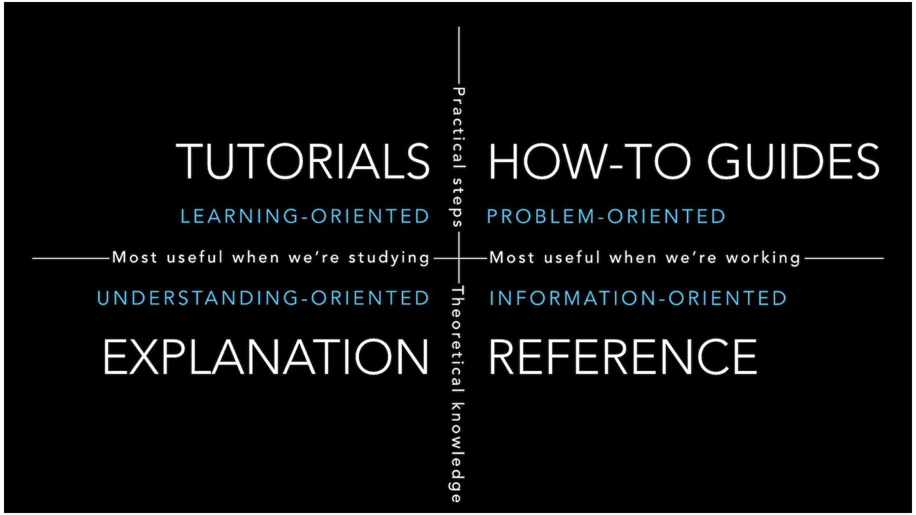
- Jupyter notebook cells can be executed in serial with the ‘Run All Cells’
- Cells can be either code or markdown.
- Documenting process next to code is easy and can be supported by table outputs, graphs and even external images for explanation.



Part 2: extensions

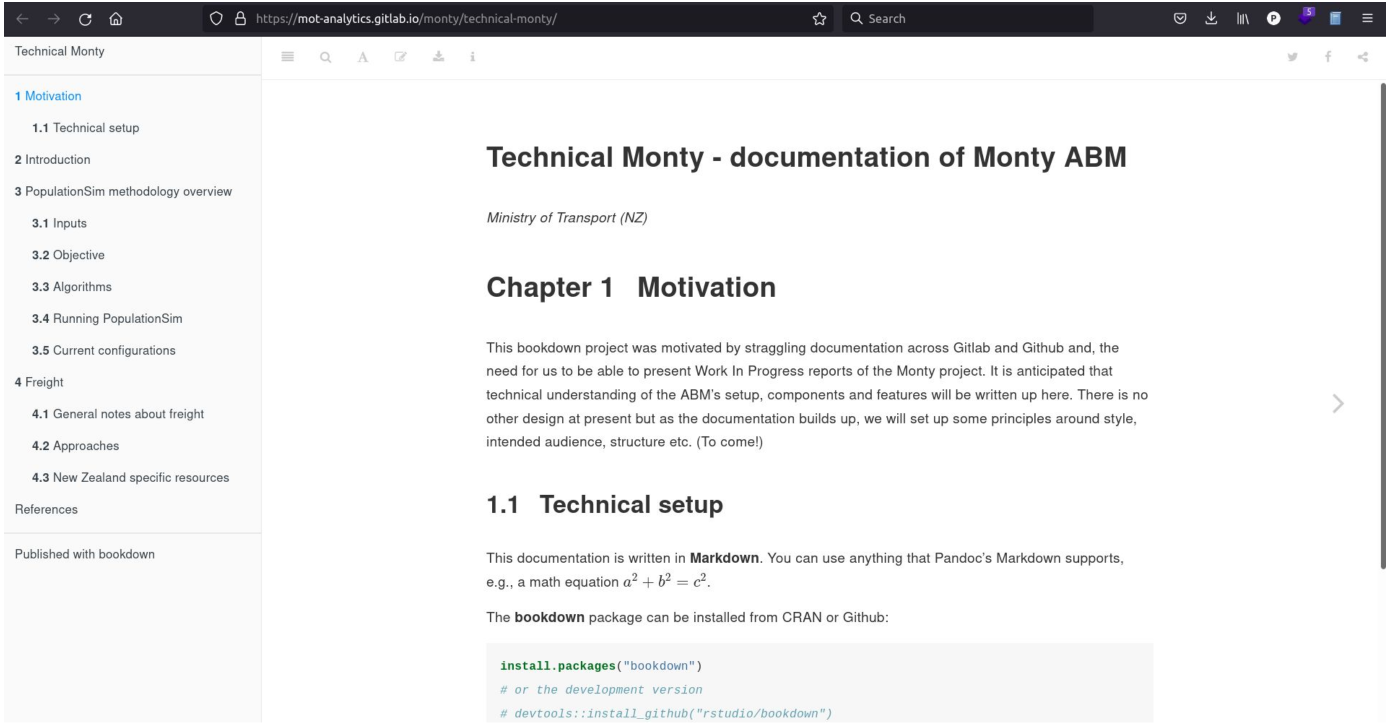
Good code *and* documentation

- Documenting alongside code helps explain the particular steps in the pipeline
- But, good documentation exists for several use-cases
- A 'website' that contains all the relevant documentation explaining process, setup, tutorials and how-tos can be helpful.



[The documentation system — Documentation system documentation (divio.com)]

Git(hub/lab) Pages with bookdown for documentation


https://mot-analytics.gitlab.io/monty/technical-monty/

Technical Monty

Published with bookdown

Technical Monty - documentation of Monty ABM

Ministry of Transport (NZ)

Chapter 1 Motivation

This bookdown project was motivated by straggling documentation across Gitlab and Github and, the need for us to be able to present Work In Progress reports of the Monty project. It is anticipated that technical understanding of the ABM's setup, components and features will be written up here. There is no other design at present but as the documentation builds up, we will set up some principles around style, intended audience, structure etc. (To come!)

1.1 Technical setup

This documentation is written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

[Home | Bookdown](#)
[GitLab Pages | GitLab](#)
[GitHub Pages | Websites for you and your projects, hosted directly from your GitHub repository. Just edit, push, and your changes are live.](#)

Creating applications with docker

¯_(ツ)_/¯

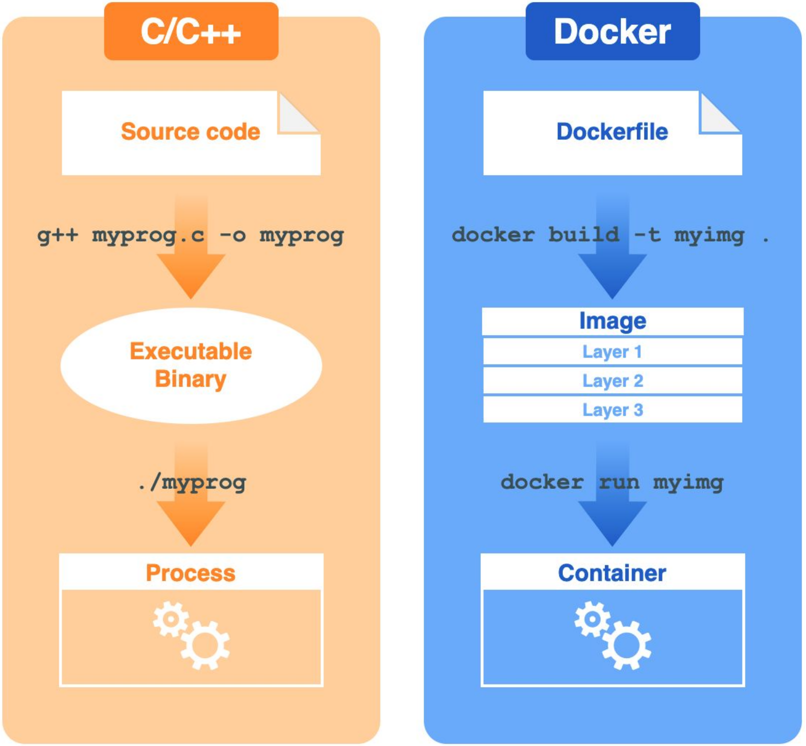
IT WORKS
ON MY MACHINE

- Images are
  - described by a Dockerfile
  - can be built on demand
  - stored in a common registry e.g. docker, gitlab, Amazon ECR
- Containers can run:
  - IDEs
  - Scripts
  - Services (like a database)

[cmhh/dockershowcase: A set of Dockerfiles and Docker Compose configurations to demonstrate various Docker use cases. (github.com)]

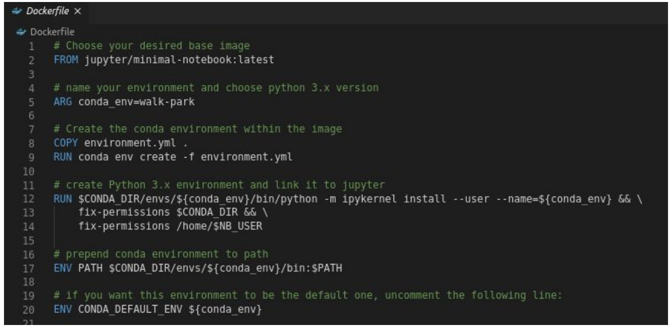
Creating applications with docker

- Like compiling source code for a programming language, creating a container also starts with a plain text file (Dockerfile)
- Similar to using a compiled binary file to launch a program, the image is then run to create a container instance.

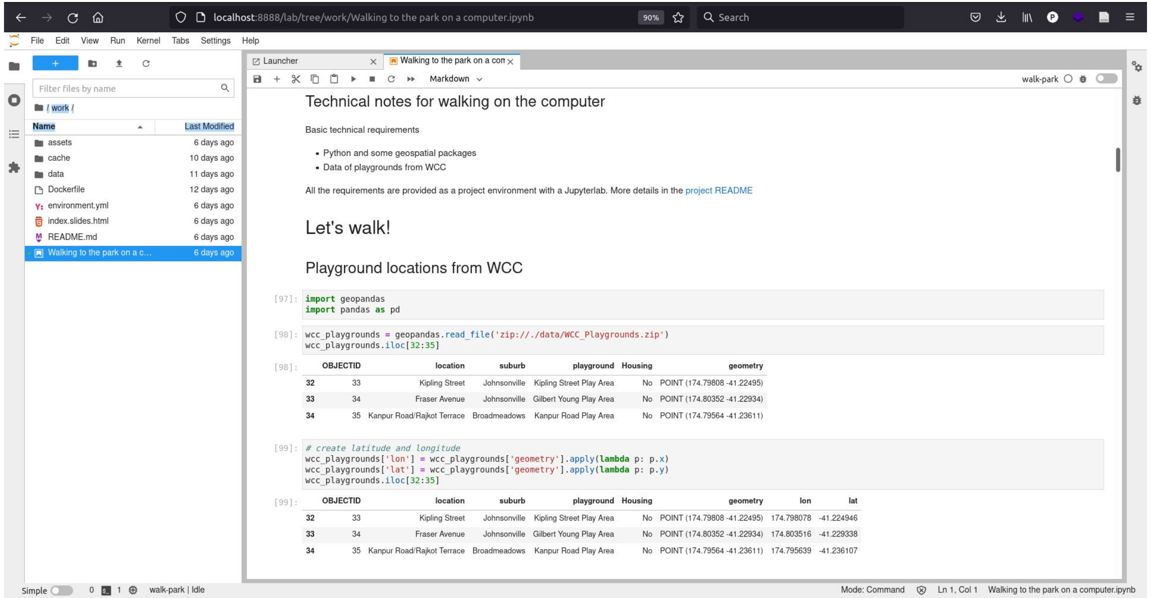


[Ten simple rules for writing Dockerfiles for reproducible data science (plos.org)]()

Dockerfile to container running an IDE



```dockerfile
# Choose your desired base image
FROM jupyter/minimal-notebook:latest

# name your environment and choose python 3.x version
ARG conda_env=walk-park

# Create the conda environment within the image
COPY environment.yml .
RUN conda env create -f environment.yml

# create Python 3.x environment and link it to jupyter
RUN $CONDA_DIR/envs/${conda_env}/bin/python -m ipykernel install --user --name=${conda_env} && \
    fix-permissions $CONDA_DIR && \
    fix-permissions /home/$NB_USER

# prepend conda environment to path
ENV PATH $CONDA_DIR/envs/${conda_env}/bin:$PATH

# if you want this environment to be the default one, uncomment the following line:
ENV CONDA_DEFAULT_ENV ${conda_env}
```



docker run -d -p 8888:8888 \
-v ${PWD}:/home/jovyan/work \
-e JUPYTER_ENABLE_LAB=yes \
-e CHOWN_HOME_OPTS='-R' \
-e CHOWN_HOME=yes \
maptime-test \
start.sh jupyter lab --LabApp.token=\'\'

Dockerfile to container running a script

- Run a script as a self-contained application
- Pass in arguments to docker run command

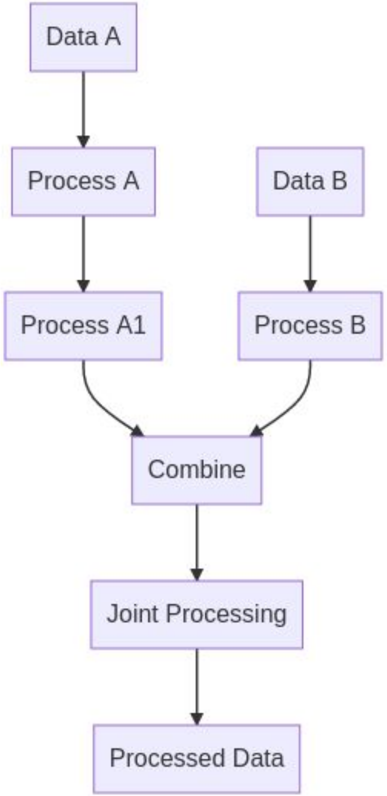
```
docker run -v ${PWD}:/tmp rail-etl \
--credentials_file='/tmp/credentials.yaml' \
--date_start='2021-06-14' \
--date_end='2021-07-15' \
--output_path='/tmp'
```

Dockerfile

```
FROM continuumio/miniconda3

WORKDIR /app

# Create the conda environment within the image
COPY environment.yml .
RUN conda env create -f environment.yml

# Make RUN commands use the new environment:
SHELL ["conda", "run", "-n", "rail-etl", "/bin/bash", "-c"]

# change? create a subfolder with app?
COPY etl /app

# # The code to run when container is started:
ENTRYPOINT ["conda", "run", "--no-capture-output", "-n", "rail-etl", "python", "etl_main.py"]

```

Linear to DAG workflows

- Directed Acyclic Graphs are very common flow structures in data processing
- Until outputs interact, task execution can be managed as parallel steps



Orchestrating DAGs

Two main approaches

- **The “Data Scientist” route**: local, often within codebase
  - kedro - Python
  - targets (formerly drake) - R
  - GNU make
  - Popper
- **The “Data Engineer” route**: cloud-deployable, scalable, schedulable
  - Airflow
  - Luigi
  - AWS Step Functions

Basic container-native workflows with popper

- A container-native task automation engine
- Runs on distinct container engines
- Orchestration framework
- Simple YAML files to describe workflows

```yaml
steps:
# download CSV file with data on global CO2 emissions
- id: download
  uses: docker://byrnedo/alpine-curl:0.1.8
  args: [-L, git.io/JUcRU, -o, global.csv]

# obtain the transpose of the global CO2 emissions table
- id: get-transpose
  uses: docker://getpopper/csvtool:2.4
  args: [transpose, global.csv, -o, global_transposed.csv]
```

https://getpopper.io

Other topics being explored

- Parquet / feather for Persistent Staging Areas (PSAs)
- SQLite for indices or data needing appends
- Rolling window processing with Parquet

Thank you!