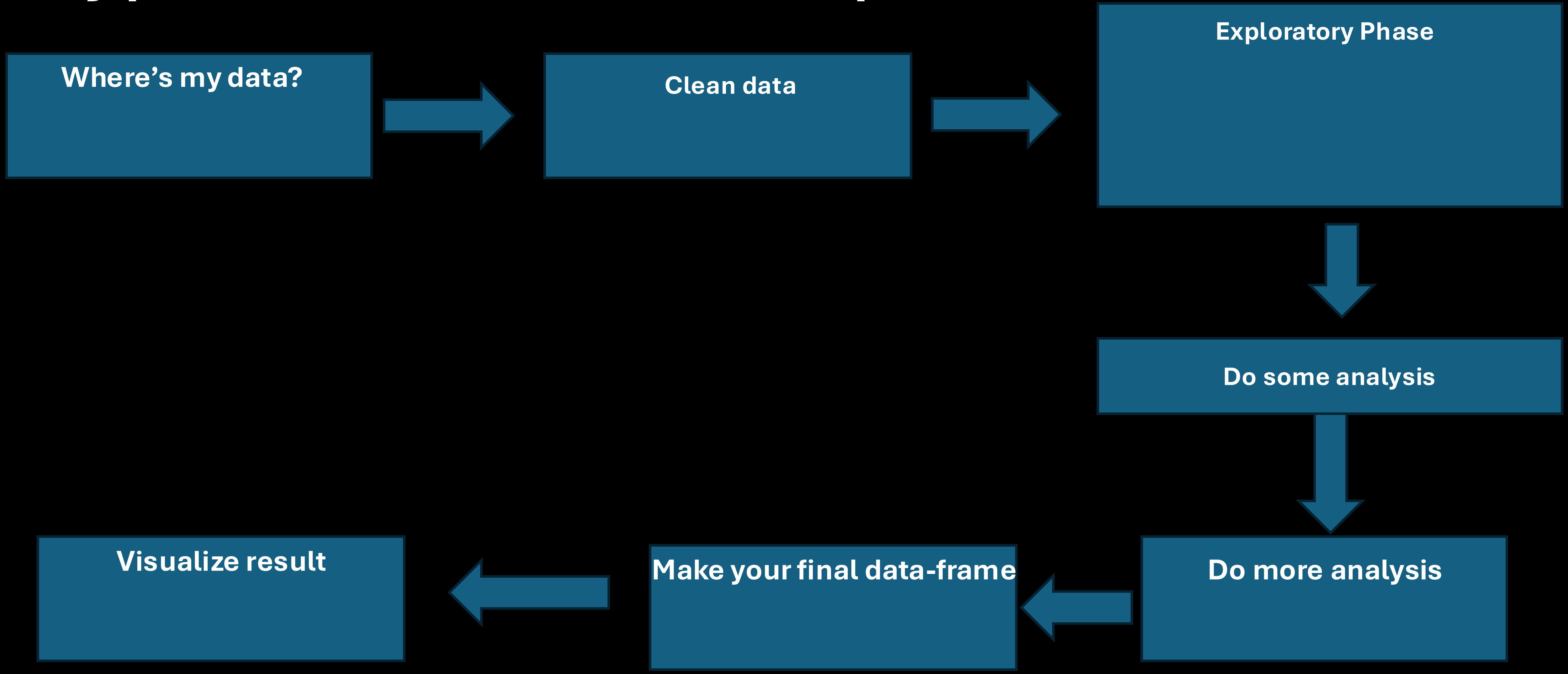
Ruffus: A Simple Python Pipeline Management Tool

Jorge Padial

AJC December 4, 2024

“Typical Scientific Compute workload”



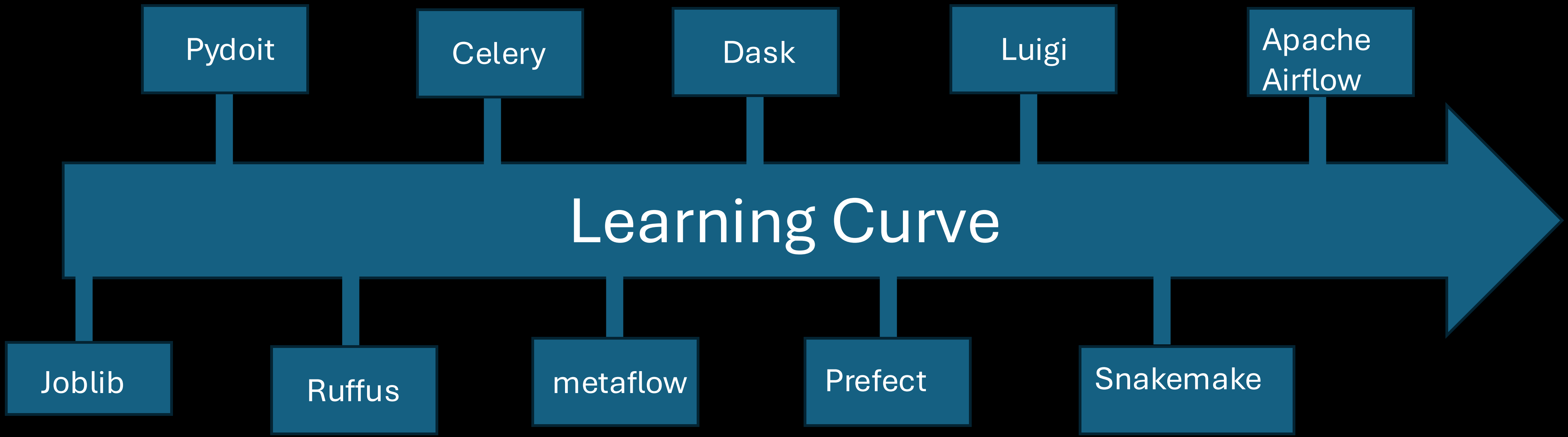
Pipeline management tools: definition

A **pipeline management tool** is a software that helps **automate**, **schedule**, and **orchestrate** the execution of a series of tasks or processes, managing dependencies, retries, and resource allocation to ensure efficient and reliable workflow execution.

Pipeline management tools: use cases

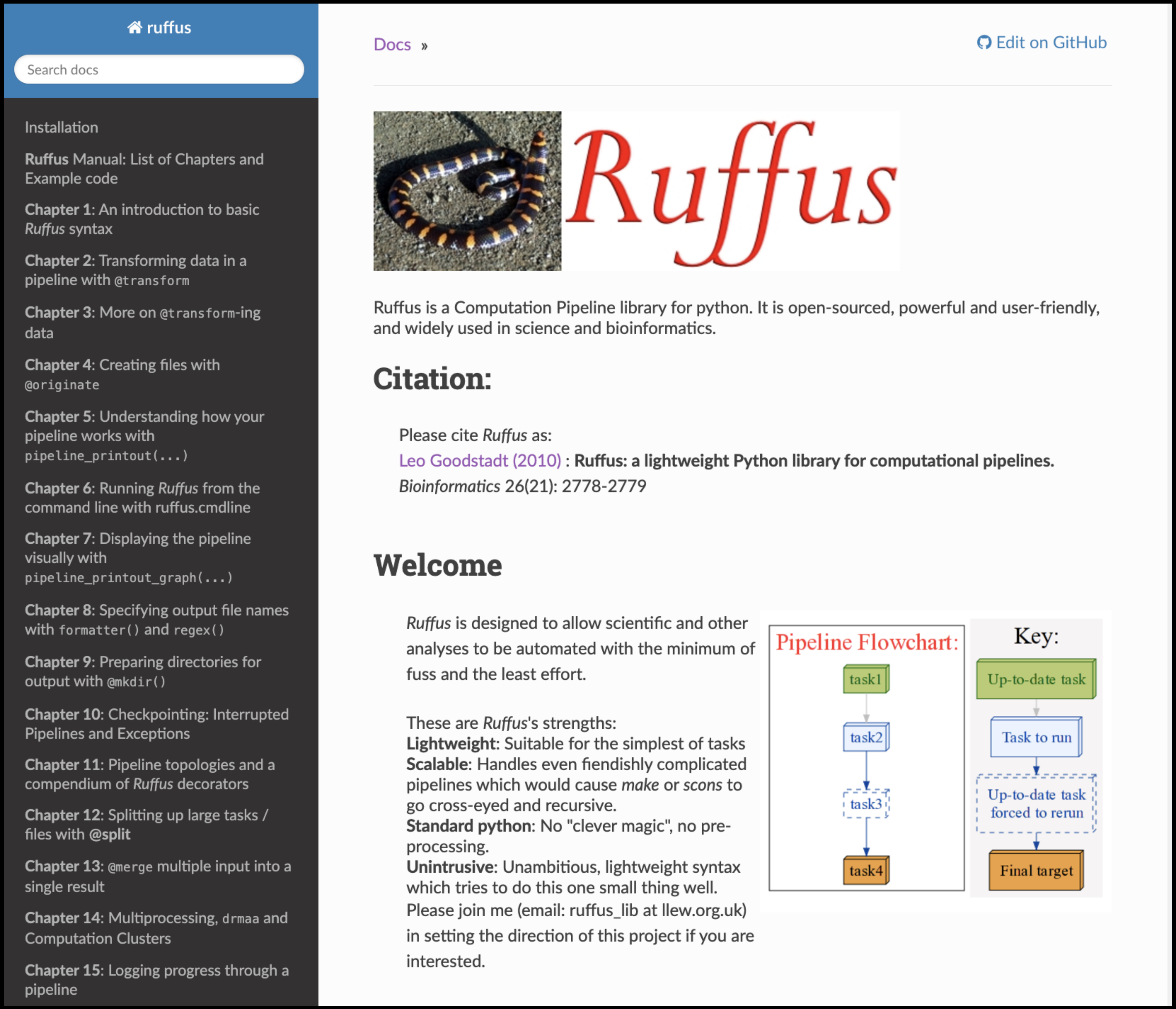
Goal	What they are	What they are not
Task Scheduling & Orchestration	- Automating and scheduling workflows. - Managing complex task dependencies with multiple steps. - Defining explicit dependencies between tasks (e.g., task A must finish before task B starts).	- Simple, single-task scripts. - Ad-hoc task execution without dependencies. - Simple, independent tasks with no need for dependency chains.
Reproducibility & Versioning	- Ensuring workflows are reproducible (e.g., storing inputs, outputs, and intermediate steps).	- One-time or non-critical tasks where reproducibility is not necessary.
Failure Handling & Retries	- Managing retries for failed tasks. - Handling error propagation and task recovery.	- Simple, non-critical tasks without failure management. - Workflows that do not need robustness or error handling.
Parallel & Distributed Computing	- Distributing tasks across multiple machines or processes. - Parallelizing independent tasks for faster execution.	- Simple sequential task processing. - Tasks that cannot be parallelized or distributed.
Data Integration & ETL	- Integrating with various data sources (e.g., databases, APIs, cloud storage).	- Workflows that don't require data integration with no need for data processing or transfer.

Pipeline management tools: examples



Ruffus

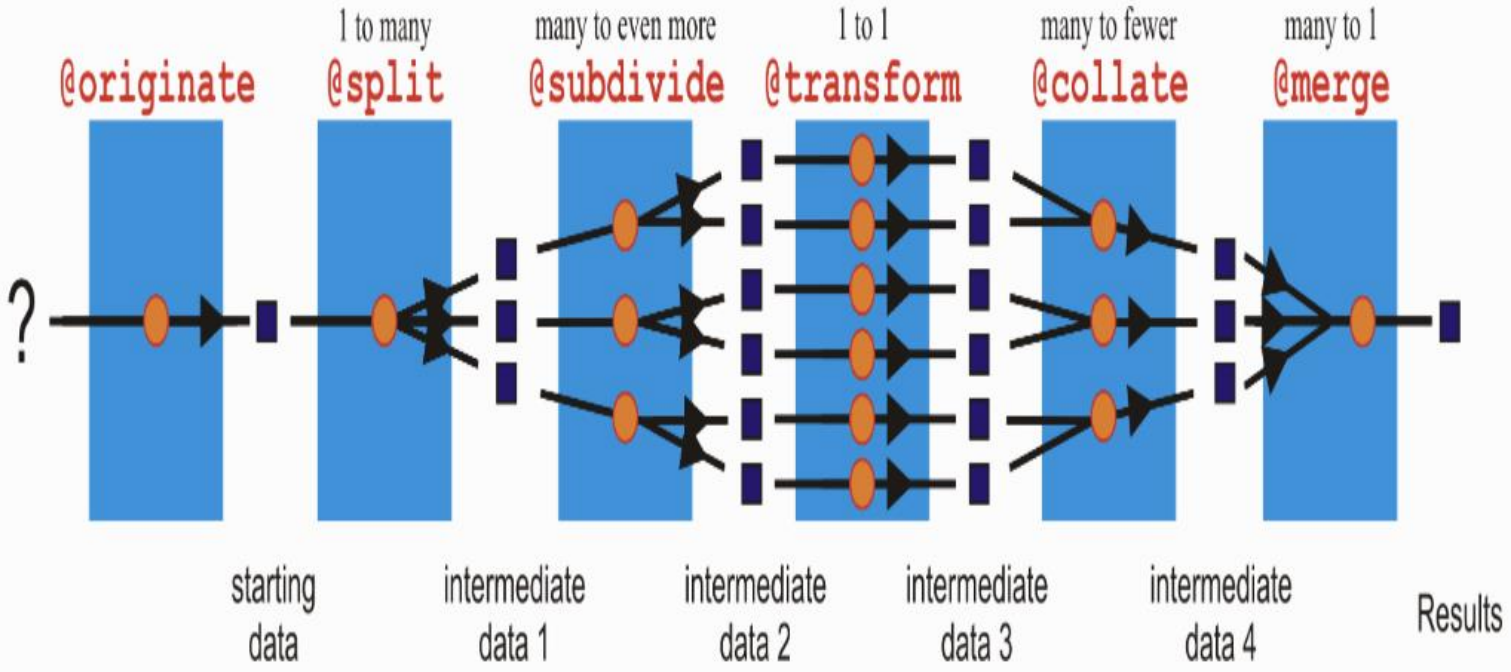
- **Simple and lightweight:**
  - Easy way to go from notebook experimentation to a full pipeline.
- **Decorator functions that map from input file(s) to output file(s).**
  - well-suited for data processing and scientific workflows
- **Easy Parallelization**
- **Platform-agnostic:**
  - Windows, Linux, Mac
- **Caches:**
  - Only runs files not up to date
- **Requires regular expressions**
- **Not easy to run on multiple nodes**



Ruffus: decorators

A bestiary of *Ruffus* decorators

Very often, we would like to transform our data in more complex ways, this is where other *Ruffus* decorators come in.

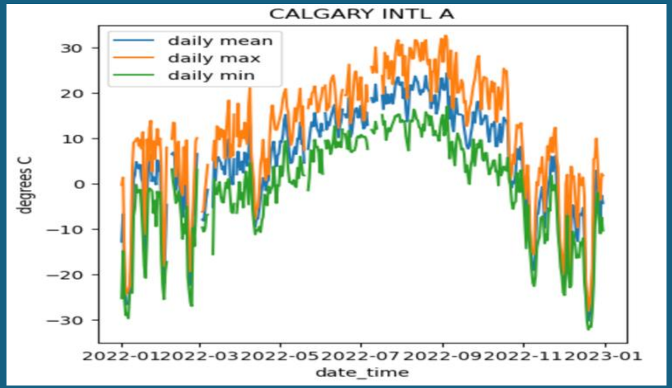


“Typical Scientific Compute workload”

Where’s my data?
Download yearly weather data from Canada

Clean data
- Standardize column names

Exploratory Phase

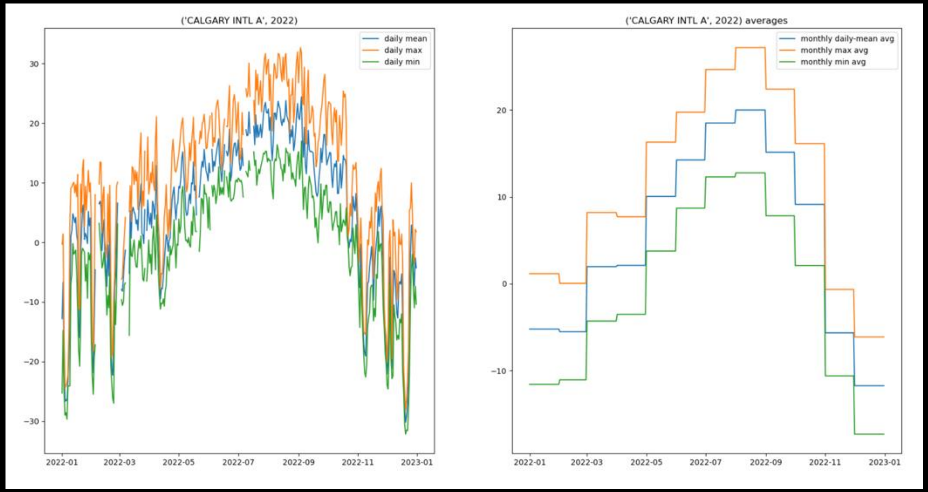


Spawn N-amount of new jobs

Do some analysis:
- Calculate mean of monthly temperatures

Concatenate N-outputs into 1 data-frame.

Visualize result:
- Plot mean monthly temperatures



Let’s view a basic Jupyter exploratory phase.

The same functions can be used to create a simple script from download to plot

```python
# Start the timer
start_time = time.time()

city_dict = {"Calgary": 50430}

# city_dict = {"Calgary": 50430,
#              "Montreal": 30165,
#             "Vancouver": 51442,
#             "Winnipeg": 51097}

result = []
list_of_years = [2022]

# create query dictionary for each city/year
for city, city_code in city_dict.items():
    for year in list_of_years:
        result.append({"City": city, "City_Code": city_code, "year": year})

# download the data
data_downloaded_list = []
for entry in result:
    data_downloaded_list.append(analysis_module.download_data(entry))


# prep the columns of the dataframe
df_list_col_prepped = [analysis_module.prep_df(this_file) for this_file in data_downloaded_list]


# do some analysis

new_df = pd.concat([analysis_module.calculate_monthly_avgs(this_df) for this_df in df_list_col_prepped]).reset_index(drop = True)


# plot data

for label, group in new_df.groupby(['station_name','year']):

    analysis_module.plot_data(label, group)

# End the timer
end_time = time.time()

# Calculate elapsed time
elapsed_time = end_time - start_time

# Print the results
print(f"Elapsed time: {elapsed_time:.6f} seconds")
```

Let’s create the same script with ruffus.

We are going to touch on 4 main things:

1. Creating a file from scratch using **@files**
2. Using **@transform** to map from 1-to-1
3. Using **@subdivide** to map from 1-to-many
4. Using **@collate** to map from many-to-1

@files: from nothing to something

- Needs a list of initialization configuration information
- In our example, the config dictionaries are the same as in a normal script. They are just called within **@files**

  - **Input file:**
    - None
  - **Output file:**
    - Calgary_2022.downloaded
  - **Config:**
    - Individual dictionary inside results = []

```python
# input list of years
list_of_years = [2022]

result = []
# Loop through the dictionary keys and list of years
for city, city_code in city_dict.items():
    for year in list_of_years:
        result.append({"City": city, "City_Code": city_code, "year": year})


# function that takes every config in result and passes it to
# the "file creator" (AKA @files decorator) in order to download
def dl_params():
    for config in result:

        infile = None

        city, year = config['City'], config['year']

        # establish the first ruffus outfile that will be created from scratch

        outfile = f'{city}_{year}.downloaded'

        yield(infile, outfile, config)



@files(dl_params) # @files will create a file from no-files made
def download_data(infile, outfile, config):

    downloaded_file_name = analysis_module.download_data(config)

    output_df = pd.DataFrame([{'downloaded_file_name': downloaded_file_name}])

    pickle.dump(output_df, open(outfile,'wb'))

if __name__ == '__main__':

    pipeline_run([download_data], verbose = 2)
```

@transfom: 1-to-1to fix the column names:

- @transform('1st arg', '2nd arg', '3rd arg')
- '1st': what was the previous step
- '2nd': what will you change from the previous functions output file.
- '3rd': what is the new extension

```python
# if __name__ == '__main__':

#     pipeline_run([download_data], verbose = 2)

@transform(download_data, suffix('downloaded'), 'fixed_columns.pickle')
def fix_columns(infile, outfile):

    infile_df = pickle.load(open(infile, 'rb'))

    csv_file_name = infile_df.iloc[0].downloaded_file_name

    prepped_df = analysis_module.prep_df(csv_file_name)

    pickle.dump(prepped_df, open(outfile, 'wb'))

if __name__ == '__main__':

    pipeline_run([fix_columns], verbose = 2)
```

- **Input file:**
    - Calgary_2022.downloaded
- **Output file:**
    - Calgary_2022.fixed_columns.pickle

@subdivide: 1-to-many split full year into months

- @subdivide('1st arg', '2nd arg', '3rd arg', '4th)
- '1st': what was the previous step
- '2nd': formatter()
- '3rd': specify files to be checked
- '4th': specify path to be created

- **Input file:**
  - Calgary_2022.fixed_columns.pickle
- **Output file:**
  - Calgary_2022.fixed_columns.#.subdivide_monthly.pickle

```python
# if __name__ == '__main__':

#     pipeline_run([fix_columns], verbose = 2)


@subdivide(fix_columns, formatter(),
            # Output parameter: Glob matches any number of output file names
            "{path[0]}/{basename[0]}.*.subdivide_monthly.pickle",
            # Extra parameter:  Append to this for output file names
            "{path[0]}/{basename[0]}")
def divide_by_month(infile, outfiles, output_file_name_root):

    infile_df = pickle.load(open(infile, 'rb'))

    print(infile)

    for month_number in infile_df.month.unique():

        copy_df = infile_df.copy()

        mask_df_for_output_by_month = copy_df[copy_df.month == month_number]

        # output_file_name_root == /path/to/where/you/are/working/{city}_{year}.fixed_columns
        # note: the output_file_name_root is the previous outfile without the extension "pickle"

        output_file_name = f'{output_file_name_root}.{month_number}.subdivide_monthly.pickle'

        print(output_file_name)

        pickle.dump(mask_df_for_output_by_month, open(output_file_name, 'wb'))

if __name__ == '__main__':

    pipeline_run([divide_by_month], verbose = 2)
```

@transform: 1-to-1 do each month individually

- @transform('1st arg', '2nd arg' , '3rd arg')
- '1st': what was the previous step
- '2nd': what will you change from the previous functions output file.
- '3rd': what is the new extension

- **Input file:** Calgary_2022.fixed_columns.#.subdivide_monthly.pickle
  Output files: Calgary_2022.fixed_columns.#.monthly_analysis.pickle

```
# if __name__ == '__main__':

#     pipeline_run([divide_by_month], verbose = 2)


@transform(divide_by_month, suffix('.subdivide_monthly.pickle'), '.monthly_analysis.pickle')
def monthly_analysis(infile, outfile):

    infile_df = pickle.load(open(infile, 'rb'))

    output_df = analysis_module.calculate_monthly_avgs(infile_df)

    pickle.dump(output_df, open(outfile, 'wb'))

if __name__ == '__main__':

    pipeline_run([monthly_analysis], verbose = 2)
```

@collate: many-to-1 join each monthly file

- @collate('1st arg', '2nd arg' , '3rd arg')
- '1st': what was the previous step
- '2nd': regex to ID all files to join.
- '3rd': what is the new extension

- **Input files:**
  - Calgary_2022.fixed_columns.#.monthly_analysis.pickle
- **Output file:**
  - Calgary_2022.joined_monthly_ananalysis.pickle

```
# if __name__ == '__main__':

#     pipeline_run([monthly_analysis], verbose = 2)


@collate(monthly_analysis, regex(r'([A-Za-z]+_\d{4}).fixed_columns.\d{0,2}.monthly_analysis.pickle'), r'\1.joined_monthly_analysis.pickle')
def join_monthly_analysis(infiles, outfile):

    joined_input_df = pd.concat([pickle.load(open(infile, 'rb')) for infile in infiles])

    sorted_df = joined_input_df.sort_values(by = 'date_time')

    pickle.dump(sorted_df, open(outfile, 'wb'))

if __name__ == '__main__':

    pipeline_run([join_monthly_analysis], verbose = 2)
```

@transfer: 1-to-1 make visualization

- @transform(‘1st arg’, ‘2nd arg’ , ‘3rd arg’)
- ‘1st’: what was the previous step
- ‘2nd’: regex to ID all files to join.
- ‘3rd’: what is the new extension

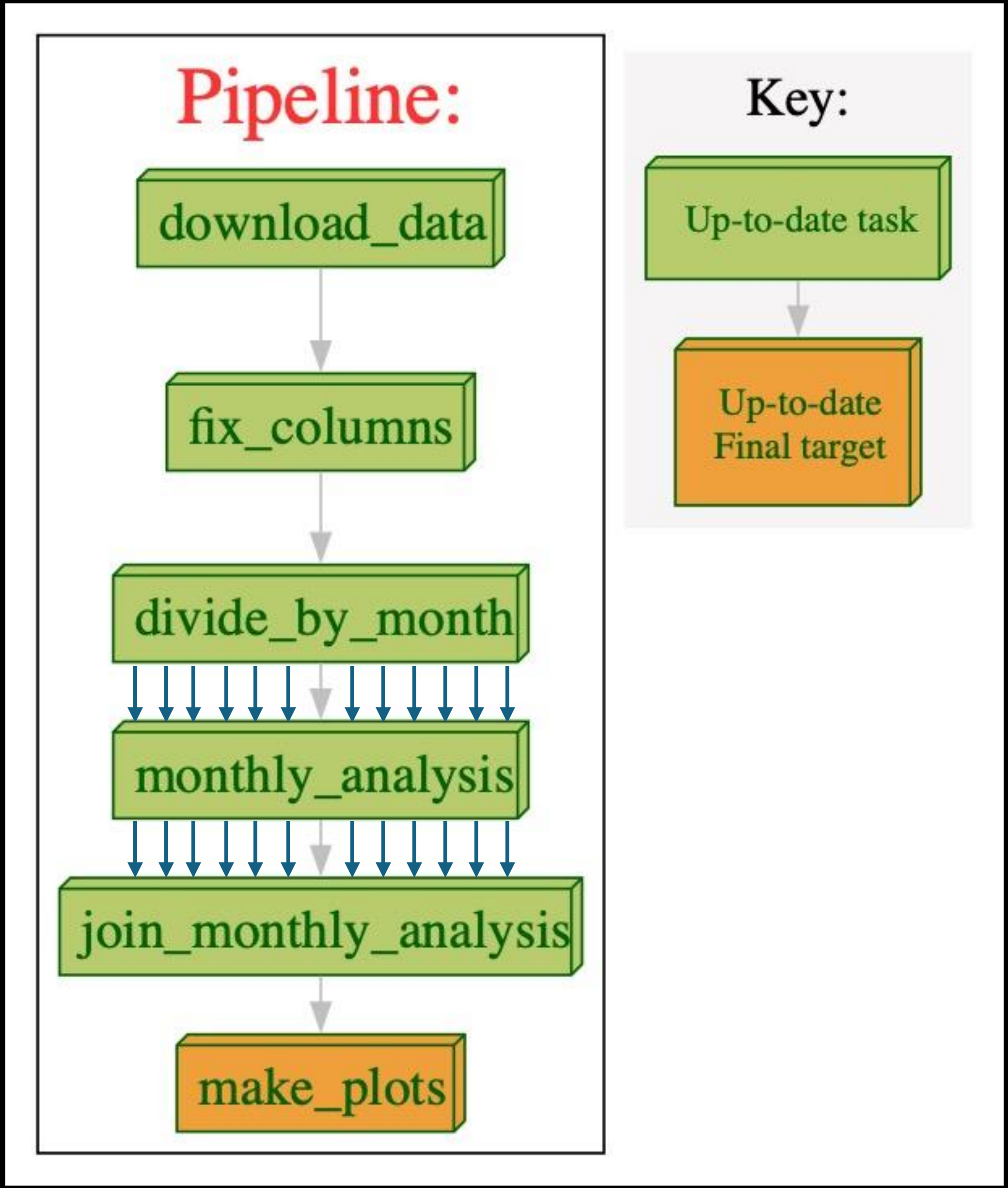
- **Input files:**
  - Calgary_2022.joined_monthly_ananalysis.pickle
- **Output file:**
  - Calgary_2022.jpg

```
# if __name__ == '__main__':

#     pipeline_run([join_monthly_analysis], verbose = 2)


@transform(join_monthly_analysis, suffix('joined_monthly_analysis.pickle'), 'plots_made.pickle')
def make_plots(infile, outfile):

    df  = pickle.load(open(infile, 'rb'))

    for label, group in df.groupby(['station_name','year']):

        analysis_module.plot_data_ruffus(label, group)

    output_df = pd.DataFrame([{'plots_made': True}])

    pickle.dump(output_df, open(outfile, 'wb'))


if __name__ == '__main__':

    pipeline_run([make_plots], verbose = 2)
```

Yay! you created your first Ruffus script!

- Everything is in 1 script!
- Every infile/outfile is cached
- Combinations of different @ operators can be made to suit your needs
- BUT:
  - ”Jorge, you said we can multi-process”

```
if __name__ == '__main__':
    pipeline_run([make_plots], verbose = 2, multiprocess= 8)
```



Links to resources

Ruffus:
https://ruffus.readthedocs.io/en/latest/

Regular expression builder:
https://regex101.com/