Day 2 - Guidelines

> Today's main topic is about configuration. First, you will learn how to build a robust and flexible config manager. Then, you will leverage this flexible config manager to build a training and a prediction system.

✔️ Prerequisites

- Be sure that the `day_1` directory is complete (you can copy-paste from our suggestions if necessary)
- **Copy all the content of `day_1` directory to a new `day_2` directory: today, we will work exclusively in `day_2`**
- Update your package installation by running `pip install -e .` **from the `day_2` directory.**

💡 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.
- Use IPython or a notebook to debug your code.
- We don't force you to use Python type annotations, as it may be too much to focus on. However, **feel free to type-annotate any function you want**.

0. Create custom errors

`Duration: 5 min`

As discussed, raising custom errors in your code can simplify the debugging process. It will give you obvious hints about what went wrong.

In `src/exceptions.py` implement the following classes if you don't have them already:

```
# src/exceptions.py
class ACMEError(Exception):
    """Base exception class for all custom errors"""

class DatasetNotFoundError(ACMEError):
    """The dataset was not found on disk"""

class ModelUnknownError(ACMEError):
    """The requested model does not exist in the model registry"""

... # + add any custom error you want.
```

1. Config re-design, part 1

`Duration: 15 min`

The `config` object from yesterday is OK, but we could do better. Currently, we are mixing within the same object:

- static config (mostly directory paths) that will hardly ever change,
- feature data and model hyper-parameters that could change more often, and have a totally different point of focus.

> This is a bad *Separation of Concerns*, and we're going to improve it right now.

1.1 Create a new `config` package

`config` was a module, we will now make it a package, for better flexibility.

Create a `src/config` directory, and make this directory a Python package by adding an empty `__init__.py` file. Add 2 other Python modules, as listed below:

▼ `config` *Toggle to view the content of* `src/config` *directory.*

- `__init__.py` *empty for now*
- `config.py` *same content as the old* `src/config.py` *(will be updated later)*
- `directories.py` *empty for now*

> 💡 **Tip**
>
> You can use the `git mv` command to automatically move your `src/config.py` file into `src/config/config.py`. docs

1.2 Create a `directories` object

This object will hold all the config that deals with directory and paths.

Move every attributes related to path or directory from `_Config` to a new `_Directories` class. The `__init__` method of this class **must take care of creating any missing directory.** Have a look at the `pathlib` documentation if you need to.

```
# src/config/directories.py

from pathlib import Path

# TODO:
#   1. Add all useful directories as instance attributes
#   2. Create any missing directory

# The following are just suggestions. Update to your specific
# files hierarchy.

class _Directories:

    def __init__(self):
  self.project_root = Path(__file__).parents[2].resolve()
        self.config = self.project_root / 'config'
        self.package = self.project_root / 'src'
        self.data = self.project_root / 'data'
        self.raw_data = self.data / 'raw'
        self.intermediate_data = self.data / 'intermediate'

  for dir_path in vars(self).values():
        # TODO: it may be a good idea to try/except here...
            dir_path.mkdir(exist_ok=True, parents=True)

directories = _Directories()
```

1.3 Update the `config` object

Directories are now handled in their dedicated object. Your `config` object just need to hold the data/model/feature configuration. Here is an example of the list of config instance attributes:

- `target`
- `date_col`
- `test_cutoff`
- `features`
- `model`
- ... ?

1.4 Refactor your program

You just made some minor changes that must be "echoed" in all the places where the config is used. If your implementation is robust, the refactoring should be very minor.

To reduce the required changes, make the `config` object directly available from the `config` package.

```
# src/config/__init__.py
from .config import config  # relative import is acceptable here
```

Now you only need to update all the places where directories are used. If you followed yesterday's guidelines, directories must be used in only 3 modules:

- `io.py`
- `run_train.py`
- `run_dataset.py`

→ Update those files so they can work with the new `config` package.

> ⚠️ **Warning**
>
> You should not have a `src/config.py` file anymore. If it is still present, **delete it**.

---

2. Config re-design, part 2

`Duration: 20-30 min`

Our current config management does the job, but will quickly show its weaknesses.

- Why are we mixing business logic (the **how**) with purely declarative/static information (the **what**)?
- Why do we need to read/write Python code to add or update plain text data ?
- How can we easily handle multiple environments, like *dev* and *production* ?

> It's time to implement a declarative configuration management system.

NOTE

Our choice is to use YAML, as it's a widely used standard to describe software configuration. We could also have chosen JSON, or TOML, or good ol' INI files... For this project, this choice if of minor importance.

2.1 Install a YAML parser library

The `yaml` library is not included in the standard library. You need to **add it to your project dependency list**, and to **install it** in your current environment.

```
# day_2/requirements.txt
...
pyyaml==5.1.2
```

```
# in your terminal, run:
pip install pyyaml==5.1.2
# OR
# you could also run this command, but the other deps are already installed
pip install -r day_2/requirements.txt
```

2.2 Creating a declarative config file

At the **root of the project** (*ie*, the parent directory of `day_2`) create a new `config` directory. Inside this new directory, create a `config.yml` that just reproduce your current Python config, in YAML.

It should be similar to this:

```
# {{ project root }}/config/config.yml
target: 'nb_sold_pieces'
date_col: 'period'
test_cutoff: '06-02-2018'
features:
  lags:
    - 1
    - 2
    - 3
  windows:
    - 3
    - 4
    - 5
model:
  name: 'RandomForest'
  params:
    n_estimators: 100
```

2.3 Update the `directories.py` module

You now have a new directory to take into account... To keep things neat and clean, add this directory location in your `_Directory` class.

```
# src/config/directories.py

class _Directories:
    # TODO: Update instance attributes to have access
    # to the new YAML config directory
  ...
```

2.4 Update the `config.py` module

Now that your config is declared as a separate YAML file, our `config.py` module just needs to parse this YAML, and return a `config` object based on the YAML content.

> This new `config` object seems like a good candidate for a `dataclass` implementation...

Delete everything in `config.py` and write the new logic.

⚠️ This new implementation must be **as transparent as possible** for the rest of your app.

```
# src/config/config.py
from dataclasses import dataclass

import yaml

# Relative imports in current subpackage are generally OK.
# You can also choose absolute import.
from .directories import directories

@dataclass(frozen=True)  # frozen => must not change after creation
class Config:
    target: str
    date_col: str
    test_cutoff: str
    features: dict
    model: dict

def load_config_file(path):
    return yaml.load(Path(path).read_text(), Loader=yaml.FullLoader)

def get_config():
    # TODO: using `load_config_file`, return a `Config` instance
    # Don't forget your `directories` object, it may be useful here!
    pass

config = get_config()
```

3. Flexibility please!

`Duration: 15-20 min`

We want the ability to override our default configuration with specific parameters depending on the environment. A typical use case would be to make the inference or training processing **faster during debugging**, when we don't really care about evaluation metrics values.

To simulate this use case, let's say **we want a specific *dev* config, where the `RandomForestRegressor` only uses 10 estimators**, instead of the 100 currently used "in production". Every other config parameter must remain identical.

Our config files must respect the DRY principle as much as possible, hence copying `config.yml` into `dev.yml` with only 1 line difference **is not a viable solution**. It would force us to do multiple file updates when we want to change the base configuration, which is highly error-prone.

> We want to build our `config` object from multiple sources, and merge those sources together.

3.1 Creating a `dev.yml` file

We will now create a config file to be used specifically during development.

Any configuration parameter that is not declared in `dev.yml` implicitly means that this parameter is the same as the one declared in `config.yml`. If a config parameter is declared both in `dev.yml` and `config.yml` the **dev config must have the priority.**

```
# {{ project root }}/config/dev.yml
model:
  params:
    n_estimators: 10
```

This YAML file means: *the config is identical for both `dev.yml` and `config.yml` **except** for the number of estimators used by the model.*

3.2 A deep merging algorithm

Now that we have 2 config files, we need a way to merge them together into a single `config` object.

> Merging several dictionaries into one is an interesting challenge! However, we don't want you to spend too much time figuring out how to do it. We suggest that you just copy/paste this code snippet.

Create a `utils` package within `src`, containing a `deepmerge.py` module. Copy the following content:

```
# src/utils/deepmerge.py
from collections.abc import Mapping
from copy import deepcopy
from functools import reduce
from typing import Optional, Any


def deepmerge(*sources: Mapping, destination: Optional[dict] = None) -> dict:
    """Merge all `sources` mappings into a single `destination` mapping."""
    destination = {} if destination is None else destination.copy()
    return reduce(_do_merge, sources, destination)


def _do_merge(destination: dict, source: dict) -> dict:
    """
    Do the actual merging between 2 dictionaries.
```

```
    If dict value is a dict itself, merge recursively.
    """
    for key, value in source.items():
        if key in destination and _is_recursive_merge(destination[key], value):
            _do_merge(destination[key], value)
            continue
        destination[key] = deepcopy(value)

    return destination


def _is_recursive_merge(x: Any, y: Any) -> bool:
    return isinstance(x, Mapping) and isinstance(y, Mapping)
```

Check if you understand every lines of this snippet.

It's a very small file, but it includes **lots of concepts**: type annotations, variable arguments, optional arguments, keyword-only arguments, recursion, nested mutations, ternary operator, reduce, usage of Abstract Base Classes, deep copy... Definitely advanced Python! 🐍

Note that it only uses objects from the standard library.

3.3 Update `get_config`

You must now update the `get_config` function so it makes good use of our deep merge algorithm.

By default, `PyYaml` automatically transform a YAML content into a Python dictionary.

To use the `deepmerge` function, you need to feed it with multiple dictionaries, and it will return **all the dictionaries values merged into a single dict**. You can test the `deepmerge` function interactively in IPython or in a notebook, to check that we're not joking 😉

```
# src/config/config.py

...

from src.utils.deepmerge import deepmerge

...

def get_config() -> Config:
    # TODO: you must update this function.
    #
    # On top of the existing `config.yml`, it must also
    # parse `dev.yml`, and merge the 2 resulting dict into
    # a single one.
    #
    # This single dict is then used to build the Config object.
```

```
    ...
    configs = # TODO, an iterable holding 2 config dictionaries
    config = deepmerge(*configs)
    return Config(**config)

...

config = get_config()
```

In a notebook or IPython, check that your new config works as expected. Remember that the **dev config must have the priority**. Hence, by default, your config should use only 10 estimators.

⏰ It goes without saying, but by now, you must have `git commit` your work at least 3 times. Maybe more.

---

4. Contextual configuration

`Duration: 30 min`

We now have a flexible config, and a way to choose one config file over the other. We have a little problem though: **this choice is hard-coded in `config.py`**. Too much coupling! 😱

We want the ability to choose one config or the other dynamically, directly from the CLI.

To that end, we will add a new abstraction: a **context** object. This context will be accessible from anywhere in the code base (as the `config` object is right now).

4.1 Create a `context` object

Our `context` is a very simple object, that could be modeled with a simple `dict`. However, it feels more convenient to access its data via attribute access (*ie* `context.my_attr`) than by key access (*ie* `context['my_attr']`).

Using a `SimpleNamespace` seems like a nicer approach. If you prefer using a plain `dict`, please do 🙂

> 💡 If you need some explanations on `SimpleNamespace`, check this StackOverflow thread. It's really simple and useful for basic use cases like this one.

Our context will be accessible via a new `src/context.py` module. For the moment, it will only hold a single `dirs` attribute, holding our `directories` object. Later on, we will dynamically attach new attributes to this `context`.

```
# src/context.py
from types import SimpleNamespace

from src.config.directories import directories

context = SimpleNamespace(dirs=directories)

# Now, all the directories are accessible via `context.dirs`
```

4.2 Create a `production.yml` config file

Now that we have a contextual object, we want to be able to use different contexts!

We want to be able to switch between 2 environments:

- dev (or debug)
- production

> `production.yml` is identical to `dev.yml`, except that our production environment requires the Random Forest model to use 300 estimators, instead of 10.

4.3 Update the `config.py` module

Once again, we need to update our config, for the better!

The `get_config` function must now take a single `context` argument. The YAML files used to build the config object must be dependant on the environment:

- If the environment is "dev", we want to load `dev.yml`
- if the environment is "production" we want to load `production.yml`

Hint Consider that the environment will be accessible at `context.environment`

```
# src/config/config.py

...

def get_config(context) -> Config:  # this is new!
    # TODO: given the context, load the appropriate config files,
    # and merge them together.

    # The config object is built either from:
    #    - config.yml + dev.yml  (for the "dev" environment)
    #    - config.yml + production.yml (for the "production" environment)

    # NOTE:
    # -----
    # The environment value is accessible at `context.environment`

    ...
    return config

...
config = get_config()  # This is now impossible! Remove this line.
```

Don't forget to update your package-level imports:

```
# src/config/__init__.py
from .config import config   # this doesn't work anymore
from .config import get_config  # optional: you can also leave this file empty
```

4.4 Update the CLI

Here comes the missing parts! The CLI should give the user the possibility to trigger one environment or the other. Based on the CLI arguments, we will enrich the `context` object, so that the environment can be accessed from anywhere.

Your have 3 tasks.

1. Add an argument called `environment` to the argparser you created in `src/cli.py`. Have a look at yesterday's slides if necessary, or check the official docs.

2. Dynamically attach this environment to the `context` object (cf. code snippet)
3. Instantiate the `config` object, and dynamically attach it to the context (cf. code snippet)

```
# src/cli.py
...
from src.config import get_config
from src.context import context
...
parser.add_argument(...)
# TODO:
# 1. add an optional argument to your argparser.

# This argument must be assignable using the `--environment` flag,
# or its short form `-e`
#
# The value of the `environment` arg can be either "dev" or "production".
# The DEFAULT value must be "dev"

...


# TODO:
# 2 & 3. Attach all the CLI arguments to your context object.
#  That way, you can access, eg, the `environment` value anywhere
#  in your code. Check the code snippet below

args = parse_cli()

for k, v in vars(args):  # 2.
    # Dynamically assign the attribute 'k' to context, with the value 'v'
    setattr(context, k, v)

context.config = get_config(context)  # 3.

...
```

Check that your new CLI works by launching a dummy run, *eg:*

```
# From a terminal:
python -m src dataset --environment production
python -m src dataset -e production  # same
python -m src dataset -e dev  # default value: not required
```

4.5 Update your code base

> The `config` object must now be accessible **only via the `context`** object, *ie* `context.config`

You must refactor every modules of your code base that use the config:

- Imports must now import `context`, ~~instead of~~ ~~`config`~~
- Function bodies have access to the config using `context.config`

This one is easy, you don't need code samples! 😄

5. Build a training system

`Duration: 15-20 min`

> The goal of this exercise is to track your training runs, and to store training artefacts to make them reusable.

For each run of the model training pipeline, 2 artifacts will be stored on your disk:

1. A serialized model,
2. A JSON file containing the associated evaluation metrics.

> 📎 **NOTE**
>
> On a real-world project, you would use a dedicated tracking tool, such as MLFlow. This is the topic for another training 😉

5.1 Create an `artefacts` directory

Within `src/data`, beside the existing `raw` and `intermediate`, **create a new directory** called `artefacts`

5.2 Update the `io` module

In `src/io.py` we need to create 3 new functions, that we will use to store the model training artefacts.

- `save_model`
- `save_metrics`
- `save_training_output` will be responsible for calling both functions (Separation of Concerns... remember? 😃)

We suggest using `joblib` to store the serialized model. It ships automatically with `sklearn`, so it's not a new dependency for our project, and it can be more efficient than the standard `pickle` module. Cf. this great SO answer.

Both `joblib` and `pickle` use the same interface to dump/load objects.

```
# src/io.py
import joblib  # or pickle.
...

# IT WOULD BE A GOOD IDEA TO USE SOME
# LOGGING AS WELL!

def save_model(model, *, path):
    return joblib.dump(model, path)

def save_metrics(metrics, *, path):
    with open(path, 'w') as f:
        json.dump(metrics, f, indent=2)  # must be human-readable

def save_training_output(output, *, directory):
    # 'output' is a directory with 2 keys: 'model' and 'metrics'
    # TODO: save those 2 artefacts on disk.
    # They must be store in files called respectively
    # 'model.joblib' and 'metrics.json'
    pass
```

5.3 Update the training pipeline

Update the `main` function of `src/run_train.py` so it can store the 2 artefacts in there expected locations.

```
# src/run_train.py
...

# TODO
# After the training process, store both model and evaluation metrics
```

5.4 Run a training pipeline

To check that your update works as expected, run a new training pipeline. Check if you have the 2 expected files on disk when it's done.

```
# in your terminal:
python -m src train -e production
# or the other environment... Now you have the choice! :)
```

❓ What is the biggest flaw of our current experiment tracker? How could we fix it?

---

6. Build a prediction system

`Duration: 20-30 min`

As we just did for the training system, we will now build a prediction system so we can easily make predictions

6.1 Add a new `input` directory

Our prediction system will read data from a local file, and feed this data to the `.predict()` method of the trained model.

Within the `data` directory, create a new `input` directory. Inside this directory, create a `predictions_input.json` file, with the following content:

```
[
  {
    "product_id": 1,
    "period": "2019-06-02",
    "gross_price": 31.3,
    "supplier": null
  },
  {
    "product_id": 1,
    "period": "2019-13-02",
    "gross_price": 13.68,
    "supplier": "John"
  },
  {
    "product_id": 2,
    "period": "2019-13-02",
    "gross_price": 13.68,
    "supplier": "John"
  },
  {
    "product_id": 2,
    "period": "2019-13-02",
    "gross_price": 13.68,
```

```
    "supplier": "John"
  }
]
```

6.2 Update the `io` module

We have a new input to read from... and now that we want to make prediction, we also need to *deserialize* the model.

> 📎 **NOTE**
>
> This `io` module would probably benefit from raising custom errors, in case that there is no existing model stored locally, or no prediction input file, etc... Feel free to enhance the suggested implementations!

Create 2 new functions, `load_model` and `load_predictions_data`.

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

# Don't forget to use some logging... It can ease your debugging process!

def load_model(path):
    return joblib.load(path)

def load_predictions_data(path):
    with open(path) as fp:
        return json.load(fp)
```

> 🌟 **BONUS**
>
> Implement a `save_predictions` function, to store the predictions on disk (optional).

6.3 Create a `predict` package

Create a new `src/predict` directory, with two files:

- `__init__.py` (empty)
- `main.py`

Within `src/predict/main.py` implement all the functions required to run a prediction. **Remember to implement a good Separation of Concerns**. Avoid writing functions that do 5 different things...

> 💡 Remember that you have a `src.context` module holding valuable data, such has directory paths, or contextual data from the CLI...
>
> You can consider that the file name of the predictions input will be given by a new CLI argument. That argument will be easily available in the `context` object 😃
>
> Enjoy the pleasure of having a flexible program! 🍾

```
def predict():
    # TODO: this function must be called with NO ARGUMENTS.
    # Its job is to get the prediction data from the input file,
    # then de-serialize the saved model, then run a `.predict()`
    # on this model, and finally return the prediction results.
    pass
```

⚠️ **IMPORTANT NOTE**

Depending on your feature set, you may have hit a wall now! If you remember, our feature set uses the **target** to build cyclical features. However, and by definition, the data points used for inference **do not** contain the target data!

You need to refactor the `src.features.build_feature_set` function, so it takes an optional boolean argument to use when we **don't want to use the `target` data**, like when we run predictions.

Maybe some of the features must disappear as well. And in that case, your must re-train a model before running inferences...

All of this is **suboptimal, to say the least**. Tomorrow, we'll see how to use better abstractions to make this problem (almost) disappear.

Conclusion

Always build a prediction system early. This kind of problems is better noticed early than the day before the first production release...

6.4 Create a prediction pipeline

To have a consistent behaviour, we will implement `src/run_predict.py` that will take care of calling the `predict()` function you just wrote.

Here is our suggestion (to be adapted, depending on the current state of your code base)

```
# src/run_predict.py
"""
Prediction pipeline
"""
import time
import logging

from src.prediction.predict import predict, save_predictions

logger = logging.getLogger(__name__)


def main():
    start = time.time()
    logger.info("Starting prediction job...")
    predictions = predict()
    # Maybe you don't have a 'save_predictions' function... That's OK.
    # You can implement it as a bonus :)
    save_predictions(predictions)

    run_duration = time.time() - start
    logger.info("Prediction job done.")
```

```
    logger.info(f"Prediction took {run_duration:.2f}s to execute")


if __name__ == '__main__':
    main()
```

6.5 Update the CLI

You now have a new pipeline, which means that your CLI must accept a new value for its command argument: `predict`

The prediction needs some input. Add the optional `-i` or `--input-data` argument to the argparser, that needs to be populated with name of the predictions input JSON file.

```
# src/cli.py

# TODO: update the pipeline choices: 'predict' is now a valid choice.
# When called, it will run the main function of 'src/run_predict.py'

# TODO: add an optional argument for 'predict': the input file name
...
parser.add_argument("-i", "--input-data")
...
```

7. Advanced logging [bonus]

`Duration: 5-10 min`

Our software is becoming better and better...

Sadly, we still have our basic config, that just prints its output to the console. It'would be much better to have solid logging configuration, that would log important messages to files, so we can read them asynchronously.

Here is a suggested logging configuration. Try to deconstruct it, and improve it so it suits your exact needs.

```
# src/config/logs.py

# NOTE:
# We are using the context object here... As such, the logging
# can't be configured in src/__init__.py anymore

import logging.config

from src.context import context

LOG_DATE_FORMAT = '%Y-%m-%d %H:%M:%S'
```

```
LOGGING_CONFIG = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'default': {
            'format': (
                '[%(asctime)s] %(levelname)s - %(message)s'
            ),
            'datefmt': LOG_DATE_FORMAT
        }
    },
    'handlers': {
        'stdout': {
            'class': 'logging.StreamHandler',
            'formatter': 'default',
            'level': 'DEBUG'
        },
        'file': {
            'class': 'logging.FileHandler',
            'formatter': 'default',
            'level': 'INFO',
            'filename': f'{context.dirs.logs}/acme.log',  # TODO: add this directory to the _Directories class
        },
    },
    'loggers': {
        'src': {
            'handlers': ['stdout', 'file'],
            'level': 'DEBUG',
            'propagate': True
        },
        '': {  # default logger, used for, eg, 3rd-party lib
            'handlers': ['stdout'],
            'level': 'WARNING',
            'propagate': True,
        },
    },
}

logging.config.dictConfig(LOGGING_CONFIG)
```

So the Python interpreter parses this file, you must ensure that this module is imported as early as possible. For example, you can tell Python to parse the `logs.py` file as soon as some object is imported from `config`.

```
# src/config/__init__.py
...
from .logs import *
```

You did it!

> This was a **very dense** program, but you've learnt **A LOT** today. Congratulations!