

Logging setup FP project



You are a Python expert helping me setup my new project.

I want it to be based on Functional Programming principles, so I will be dividing my application into three segments:

- impure: directory with impure functions
- pure: directory with pure functions
- main.py: executable file that will coordinate pure and impure functions

Project theme: the goal of the project is creating a Monte Carlo simulation to find out the most optimized portfolio containing a set of Dow Jones companies.

I will give more details about the project as I go.

Right now help me setup logging as you would do for a professional application.
Where should I put the logging setup? What configurations should I use?

Great, Gabriel. Since you're aiming for a professional-quality Python application with a clear functional separation, your logging setup should be:

1. **Reusable across the project.**
2. **Configurable for different environments (development, testing, production).**
3. **Separable from the simulation logic (pure/impure).**

📁 Recommended File Structure (early stage)

```
bash

├── impure/
│   ├── __init__.py
│   └── io_utils.py          # For file/database/network IO
├── pure/
│   ├── __init__.py
│   └── monte_carlo.py      # Pure simulation logic
├── main.py
├── logging_config.py       # Central logging setup
└── config/
    └── logging.yaml        # Optional: for external configuration
```

✅ Step 1: logging_config.py

```
python

import logging
import logging.config
from pathlib import Path
```

```
import yaml

def setup_logging(default_path='config/logging.yaml', default_level=logging.INFO):
    """Setup logging configuration"""
    path = Path(default_path)
    if path.is_file():
        with open(path, 'r') as f:
            config = yaml.safe_load(f.read())
            logging.config.dictConfig(config)
    else:
        logging.basicConfig(level=default_level)
```

✓ Step 2: config/logging.yaml (external config file - optional but cleaner)

```
yaml

version: 1
disable_existing_loggers: False

formatters:
  standard:
    format: "%(asctime)s [%(levelname)s] %(name)s: %(message)s"

handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: standard
    stream: ext://sys.stdout

  file:
    class: logging.FileHandler
    level: INFO
    formatter: standard
    filename: logs/app.log
    encoding: utf8

loggers:
  myapp:
    level: DEBUG
    handlers: [console, file]
    propagate: no

root:
  level: INFO
  handlers: [console, file]
```

Create the `logs/` directory beforehand, or modify the handler to rotate or ignore missing paths.

✓ Step 3: Use it in main.py

```
python

from logging_config import setup_logging

setup_logging()

import logging
logger = logging.getLogger(__name__)

def main():
    logger.info("Starting Monte Carlo simulation...")
    # Your coordination logic here
```

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

📌 Where to log:

- Use `logger.debug()` for intermediate steps or Monte Carlo iterations.
- Use `logger.info()` for major events (start/end simulation, best result).
- Use `logger.warning()` and `logger.error()` in impure parts (e.g., failed HTTP request or missing data file).
- Avoid logging in pure functions (they should return results and be side-effect-free).

Would you like me to scaffold this structure for you in code so you can copy/paste it?



Perfect. Now I want to setup a data loader. The only issue is that I don't know exactly how I should treat errors according to Functional Programming principles.

Here is how my function for fetching tickers looks like:

```
python
def get_djia_tickers() -> list[str]:
    """Get Tickers from Dow Jones components by accessing Wikipedia article."""

    DOW_JONES_WIKI_URL = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'

    try:
        response = requests.get(DOW_JONES_WIKI_URL)
        response.raise_for_status()
    except requests.exceptions.RequestException as e:
        raise RuntimeError(
            f"Failed to fetch data from {DOW_JONES_WIKI_URL}: {e}") from e

    tables: list[pd.DataFrame] = pd.read_html(response.content.decode('utf-8'))

    for table in tables:
        if 'Company' in table and 'Exchange' in table and 'Symbol' in table:
            logger.info('Fetched information from Wikipedia succesfully')
            return list(table['Symbol'])

    raise ValueError(
        "Could not find the expected table structure in the Wikipedia page.")
```

What would the Functional Programming best practices dictate?

You're asking a great question, Gabriel. In Functional Programming (FP), we aim to:

1. **Separate pure from impure code.**
2. **Avoid exceptions as control flow.**
3. **Represent errors as values.**

💡 What's the FP way to handle this?

Rather than *raising* exceptions (which are side-effects), return an **explicit result type** that captures either success or failure.

In Python, the common ways to do this functionally:

✅ Option 1: Return a Result-like type

```
python

from typing import TypeVar, Generic, Union, NamedTuple
from dataclasses import dataclass

T = TypeVar('T')
E = TypeVar('E')

@dataclass
class Ok(Generic[T]):
    value: T

@dataclass
class Err(Generic[E]):
    error: E

Result = Union[Ok[T], Err[E]]
```

🔧 Refactored version of your function

```
python

import requests
import pandas as pd
import logging
from typing import Union

from result import Ok, Err, Result # Assuming the above definitions are in result.py

logger = logging.getLogger(__name__)

def get_djia_tickers() -> Result[list[str], str]:
    """Fetch Dow Jones tickers from Wikipedia and return them functionally."""
    url = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'

    try:
        response = requests.get(url)
        response.raise_for_status()
    except requests.exceptions.RequestException as e:
        return Err(f"Failed to fetch data from {url}: {e}")

    try:
        tables = pd.read_html(response.content.decode('utf-8'))
    except ValueError as e:
        return Err(f"Failed to parse HTML from {url}: {e}")

    for table in tables:
        if {'Company', 'Exchange', 'Symbol'}.issubset(table.columns):
            logger.info("Fetched information from Wikipedia successfully")
            return Ok(list(table['Symbol']))

    return Err("Could not find the expected table structure in the Wikipedia page.")
```

📌 Handling the result in `main.py` or elsewhere

```
python

result = get_djia_tickers()

if isinstance(result, Ok):
    tickers = result.value
    # Proceed with simulation
elif isinstance(result, Err):
    logger.error(f"Error fetching tickers: {result.error}")
    # Handle error or exit gracefully
```

🧠 Why this is FP-friendly:

- No exceptions bubbling up.
- Errors are **explicit** and handled by the caller.
- Code becomes **composable** and easier to test (especially your impure layer).
- Pure and impure parts are clearly decoupled.

If you want, I can create a full `result.py` module for you and suggest how to test `get_djia_tickers()` independently. Would that be helpful?



Help me understand the Result-like type you came up with.

There are several things I don't understand:

- Firstly: what is 'T' and 'E'? What do they stand for?
- What is a dataclass?
- What is a Generic?

Excellent questions, Gabriel! You're diving into the core ideas behind type-safe functional patterns in Python. Let's break it all down one piece at a time:

📦 1. What are T and E?

They are **type variables** used for **type parameterization**.

Meaning:

- T typically stands for "**Type of the success value**"
- E typically stands for "**Type of the error value**"

These are just conventions, like x and y in math. You could name them anything, but T and E are idiomatic.

Analogy:

Imagine you're designing a function that could return either:

- a success value of type `list[str]`, or
- an error message of type `str`.

With `T` as `list[str]` and `E` as `str`, you can build something like:

```
python
Result[list[str], str] # A Result that is either Ok(list[str]) or Err(str)
```

2. What is a `@dataclass`?

A `dataclass` is a Python decorator that **automatically generates** boilerplate code for classes that are **just containers for data**.

Features:

- Auto-generates `__init__`, `__repr__`, `__eq__`, etc.
- Clean and readable code

Example:

```
python
from dataclasses import dataclass

@dataclass
class Ok:
    value: int
```

Is equivalent to writing:

```
python
class Ok:
    def __init__(self, value: int):
        self.value = value
    def __repr__(self):
        return f"Ok(value={self.value})"
```

3. What is Generic?

Generic is used to **define parameterized types** in Python. It's what allows `Ok` and `Err` to work with **any type**, not just one specific type.

Example:

```
python

from typing import TypeVar, Generic

T = TypeVar('T')

class Box(Generic[T]):
    def __init__(self, item: T):
        self.item = item
```

Now you can use:

```
python

Box          # Box containing an int
Box[str]("hi") # Box containing a string
```

Without `Generic[T]`, Python wouldn't know how to keep track of what type `Box` contains.

Bringing it all together:

Here's what's happening in the `Result` type:

```
python

T = TypeVar('T') # Success value type
E = TypeVar('E') # Error value type

@dataclass
class Ok(Generic[T]):
    value: T

@dataclass
class Err(Generic[E]):
    error: E

Result = Union[Ok[T], Err[E]]
```

So `Result[int, str]` means:

Either `Ok(value: int)` OR `Err(error: str)`

It gives you type-safe and pattern-matchable control flow—exactly what you'd want in a functional-style app.

Would you like me to turn this into a ready-to-use `result.py` module with a docstring and examples?