Course Notes

Development Tools

- Consider adding an extension like `prettier`
- `sudo` = superuser do
- `bin` stands for binary
- Use of virtual environments (`venv`) is worth it!
- `source` is the bash utility that allows us to run files

Git Commands and Environment

Basic Git Operations

- Keep Git environments separate for different projects
- Create new branch: `git switch -c <branch-name>`
- Create empty file: `touch <filename>`
- Push new branch upstream: `git push -u origin <branch-name>`

Git PR Workflow

For a new PR, push a local branch that doesn't have a PR to upstream tracking.

Additional Git Tips:

- Use `git add -p <filename>` to select changes interactively

MkDocs

Basic Commands

- View all options: `uv run mkdocs --help` or `uv run mkdocs serve --help`
- Stop server: `Ctrl+C` (Windows and Linux) (`Ctrl+Z` hides it and then use `fg` to bring it back to foreground)
- Configuration file: `mkdocs.yml` specifies MkDocs settings

Package Management

Legacy Package Management

- `requirements.txt` was traditionally used to specify dependencies
- Installed via `pip install -r requirements.txt`
- Now replaced by modern tools like `uv lock` and `uv sync`

Semantic Versioning (e.g., 1.6.1) of modules and software

- **Patch number** (last digit): Bug fixes and security updates that shouldn't break code
- **Minor version** (middle digit): New features and functionality
- **Major version** (first digit): Breaking changes

Version Specifiers

- Exact version: `mkdocs==1.6.1`
- Version range: `mkdocs>=1.6.1,<2` (pip selects newest compatible version)
- Dependencies between packages can cause version conflicts

Testing with Pytest

Basic Usage

Run tests with pytest:

- Run all tests: `uv run pytest`
- Run tests in specific file: `uv run pytest folder/specific_test.py`
- Run specific test: `uv run pytest folder/specific_test.py::test_name`

Pytest Parametrization

- Use `import pytest` to enable parametrization
- Add `@pytest.mark.parametrize` decorator before test functions
- Example usage for testing function: `python passengers_per_day(passengers_per_year, days_per_year)`
- `@pytest.mark.parametrize` allows multiple test cases

Test Values and Best Practices

- Can use specific pre-calculated values
- Or run function once to determine expected values

Test File Naming

- Pytest won't find files named like `fleet_test.py`
- Must follow pytest naming conventions

Style Tips

- Adding trailing commas in lists enables cleaner diffs: `python`
  ```
  my_list = [
      'item1',
      'item2',
      'item3',  # Note trailing comma
  ]
  ```
- Double click failed tests in output to jump to location

Approximate Comparisons

- `pytest.approx` supports two types of tolerances:
- Absolute tolerance (fixed number)
- Relative tolerance (percentage of value, e.g., 10%)

Development Environment Issues

Cursor Dependency Issues

There are issues with Cursor when adding the right dependencies:

```
kpb30@WL-2023-CDT01:/mnt/c/Users/kpb30/Documents_github/aviation$ uv run which
python
/mnt/c/Users/kpb30/Documents_github/aviation/.venv/bin/python
```

- Need to add it to `uv`
- Reference: https://github.com/astral-sh/uv/issues/8558
- Ask Brockie why `uv run python` doesn't work for running it in WSL

Documentation with Docstrings

Overview

- Use `man` (manual) or `help` commands for documentation
- `mkdocstrings` automatically creates documentation on MkDocs from docstrings

Setting Up API Documentation

1. Create a `docs/api/` folder
2. Add files like `fleet.md` which contains: `:::aviation.fleet`
3. This creates documentation for any functions with docstrings

Module Documentation Structure

- In the main `index.md`, it takes all functions declared in aviation's `__init__.py`'s `__all__`
- `index.md` is the default name for web content (special name for `__init__.py`)
- Can provide module docstrings

Documentation Formatting and Rules

- **Note**: Ruff only does Python code formatting, no good VS Code extension for docstring alignment
- Ruff 'D' rules are documentation rules
- Tests are for developers, not for documentation, so can ignore docstring rules on tests

Docstring Styles

In the D rules, there are two main schools of docstrings:

- **Numpy style**
- **Google style** (used in AIA)

Reference: Google Style Docstring

Advanced Documentation Features

- Can include examples in docstrings with example code showing usage
- Can get pytest to run examples in docstrings

Advanced Dependencies and Authentication

MkDocs Prebuild

- `mkdocs-prebuild` is an AIA-specific documentation dependency
- Eliminates need to manually add navigation from docsite
- Not public, so need to configure `uv` for access

GitLab Authentication

Environment Setup

Check current environment variables:

```
env | grep UV
```

For now, might need to run this every time opening a new terminal:

```
source ~/.env
```

Add the dependency:

```
uv add --group=docs mkdocs-prebuild
```

GitLab Tokens and Variables

- **Variables** (`vars.UV_INDEX_GITLAB_USERNAME`): Not secret, stored in plain text
- **Secrets** (`secrets.UV_INDEX_GITLAB_PASSWORD`): Stored in encrypted form

To-Do Items

Lunch break task: Move folders to Linux and figure out how to avoid needing to run `source ~/.env`

Runtime Checks and Type Safety

Runtime Type Checking

Example scenario: What if someone tries to use functions with wrong input types?

Test-Driven Development

Write tests first, then make changes to codebase to pass tests:

```
with pytest.raises(TypeError, match="Argument `passengers_per_year` passed to
`passengers_per_day` function must be an instance of <class 'float'>"):
    passengers_per_day("365_000_000.0", 365.0)
```

Multiline Strings and F-Strings

- Can create multiline strings using parentheses
- F-strings are available for string formatting
- `isinstance` checks can get very long and verbose for testing
- Some libraries simplify this, but code checks at runtime anyway

Static Type Checking

- Can perform ahead-of-time type checking using static analysis
- Similar to linting, but focuses on code correctness
- Use `mypy` as a static type checker for Python
- **PEP 484** allows function type specification:

```
def greeting(name: str) -> str:
    return 'Hello' + name
```

Development Tips and Tools

VS Code Tips

- Use `Ctrl+D` to highlight and replace all instances of selected text

Static Type Checking

- Run `uv run mypy .` to check all files for type issues
- Configure CI pipeline (`check.yml`) to run fastest checks first for quick feedback

Ruff Configuration Changes

Change all the ruff rules from the old one

select = ["B", # flake8-bugbear "D", # docstrings "E", # pycodestyle "F", # Pyflakes "I", # isort "N", # pycodestyle "SIM", # flake8-simplify "T", # pycodestyle "UP", # pyupgrade "W291", # no trailing whitespace] ignore = ["E501"]

to now all and a lot of ignores

`S101` causes issues when any assert is used, because it can be bypassed (not robust). We want to ignore this rule for our tests directory

`INP001` is that some functions are imported without an **init** (analysis directory looks like a package) We want to ignore this rule for our analysis directory

Ignore `T201` (print statements) in all `analysis/*.py` files. Ignore `D1` (docstrings) in all `tests/*.py` files.

PT007 is using tuples or lists

Imports of modules

This represents the package name and the module.

Now we have added the functions in the fleet module to the package namespace. This means we can now import the functions directly from the package namespace with the import in `__init__.py`. Here are the different ways to import and use functions:

1. `python`
   `import aviation`
   `aviation.fleet.passengers_per_day()`
   `aviation.passengers_per_day()`
2. `python`
   `from aviation import fleet`
   `fleet.passengers_per_day()`
3. `python`
   `from aviation import passengers_per_day`
   `passengers_per_day()`
4. `python`
   `from aviation.fleet import passengers_per_day`
   `passengers_per_day()`

Decorators

Decorators are essentially equivalent to creating a higher order function

```
@transform
def function
```

is like

```
function = transform(function)
```

To get the strings of the inputs we need to use the inspect module

```
tuple(inspect.signature(function).parameters.keys())
```

- **`abc`** stands for *Abstract Base Classes* and is imported from `collections.abc`.

- A `class` defines a new type in Python.
- By convention, type names (classes) use **CapitalizedNames**.
- `Transform[R, **P]` is an example of adding type variables (generics) to a class.

Notes on Type Variables and File Organization

- For **Callables** (i.e., function types), the convention is to specify the input types first, followed by the output type.
- For other constructs (like our new `Transform` class), the output type comes first, then the parameters.

Type Variable Explanations

- `R` is a simple type variable representing the return type.
- `**P` is a special type variable that captures both positional (`args`) and keyword (`kwargs`) arguments.

File Naming and Imports

- The file is named `_model.py` (in `src/aviation/`) to remind us that it's only needed for type definitions and not required when using the actual `camia-engine`.
- There is no need to add `collections`, `inspect`, or `typing` to your dependencies, as they are all included in the standard Python distribution.

Transforms and names

the name passengers_per_day can be the name in inputs, or the value inputed, or the transform that is calculating it from other values

New class: SystemsModel(args) which takes all the transofmrs of a module (aviation.transofmrs)

Goal is to be able to run

```
systems_model = SystemsModel(aviation.transforms)
required_global_fleet = systems_model.evaluate(inputs,outputs)
```

passengers_per_day interim transform is now totally abstracted and not mentioned

tuple[Transform, ...] means that your input could be any number of transform types tuple is the most efficient data structure (memory wise), esp if you won't be changing anything about it

Sequence are homogeneous datastructure: lists, tuples/dict (can also be heterogeneous eg mixing strings and floats),

Do the type-ing first and raise not implemented error

First test cases when outputs are known

pytest fixture: is something that is needed multiple times rather than needing to instatiate 5 times

How to keep doing multiple commits: add in tests one by one if not too artificial

Python for loops can have an else branch (if we never get break)

```
code #noqa: ANN401
```

means that that rule won't be checked on that line, bad practice, but necessary for the transform as there is no predefined type

mkdocs-prebuild camia-model camia-engine are all from the aia gitlab and need token authentification

Actually only need the engine in the analysis, not the actual package

package names (camia_engine in src/dirname) are different from package identifiers (camia-engine, in pyrpoject.toml). The folder name doesn't matter at all but sometimes useful to havee

Helpful Commands

Here are some useful commands for code formatting, linting, type checking, and testing:

- **Format code with Ruff:**

```
bash
  uv run ruff format
```

- **Check code with Ruff:**

```
bash
  uv run ruff check
```

- **Type check with mypy:**

```
bash
  uv run mypy .
```

- **Run tests with pytest:**

```
bash
  uv run pytest
```

- **Run all pre-commit hooks:** `bash`
  `uv run pre-commit run --all-files` or `bash`
  `uv run pre-commit run -a`

Units

Typing and Units in `camia-model`

- The `units` module is also part of `camia-model`.
- **Always use the `typing` library** for type annotations.
- Instead of using plain `float` in function signatures, use:

  ```python import typing from camia_model.units import day, year

  def my_function(x: typing.Annotated[float, day / year]) -> None: ... ```
- To use units like `day` or `year`, import them directly: `python`
  `from camia_model.units import day, year`
- **Adding new units:** If you need a unit that isn't provided (e.g., `passenger`), create a new `units.py` and define it there.

### Internal Style Guide

- Always import the main package as:
- `import camia_model as model`
- or, for aia-model-distribution: `import aia-model-distribution as distribution`
- **Exception:** For units, always import them directly from `camia_model.units`.

### Python Function Arguments

- Use `*args` and `**kwargs` as needed, following standard Python conventions for flexible argument passing.
- Use Python's type annotation syntax for clarity and type checking.

```
class F:
   def __call__(self, *args, **kwargs):
      print(f"Calling f with {args} and {kwargs})

   def __getitem__(self, item):
      print(f"Getting item {item} from f")

   def __add__(self,other):
      print("Adding ...")

   def __radd__(self, other):
      print("Adding on the right ...")
```
```

```
f = F()

f(0,1,b=5,a=2) # returns "Calling f with (0,1) and {'b':5,'a':2}
f["Danial"] # returns "Getting item Danial from f"
f + 1 # returns "Adding ..."
1 + f # returns "Adding on the right ..."
f.__add__(1) # quicker computational time to do f+1
```

`pytest_camia` is an extension package so camia comes after It also adds some new tests on any transform we've defined to check units of our transforms. On runtime we will just operate on floats, only for testing/dvp you care about units `camia_engine` is a package fully developed by cmamia

Switching / Enums

If we have to different models for future demand e.g. linear vs exponential growth

need enum

@enum.unique makes sure all of the ones are unique enum.auto() just gives values but we don't care what it is as we will just use the names

for classes -> convention is using title class like DemandGrowthModel

enum convention is have all capitals with potential underscore EXPONENTIAL S_CURVE etc.

Unfortunately no way to get ruff/linter and mypy to understand that having two functions. So need to add the comment

```
# noqa: F811
# type: ignore[no-redef]
```

no point in match case inside a functions: different inputs and outputs and huge variation

when you run a def function with passengers_per_year.context()

How to produce a lot of results

Transforms are able to do full arrays (fully vectorised)

can change inputs then to camia.model set input "demand_growth_model": model.AssociativeTensor([4.0 * percent/year, 2.5*percent/year], dims= ("demand_growth_model",), coords=(demand_growth_model,),)

vector results: dims has the names of dimensions coords are numbers