Introduction to Python Package Development

1. Writing documentation to a function and package
2. Import other python files in the package
3. Create a package and make it installable and downloadable
4. Testing the package

Proof of concepts

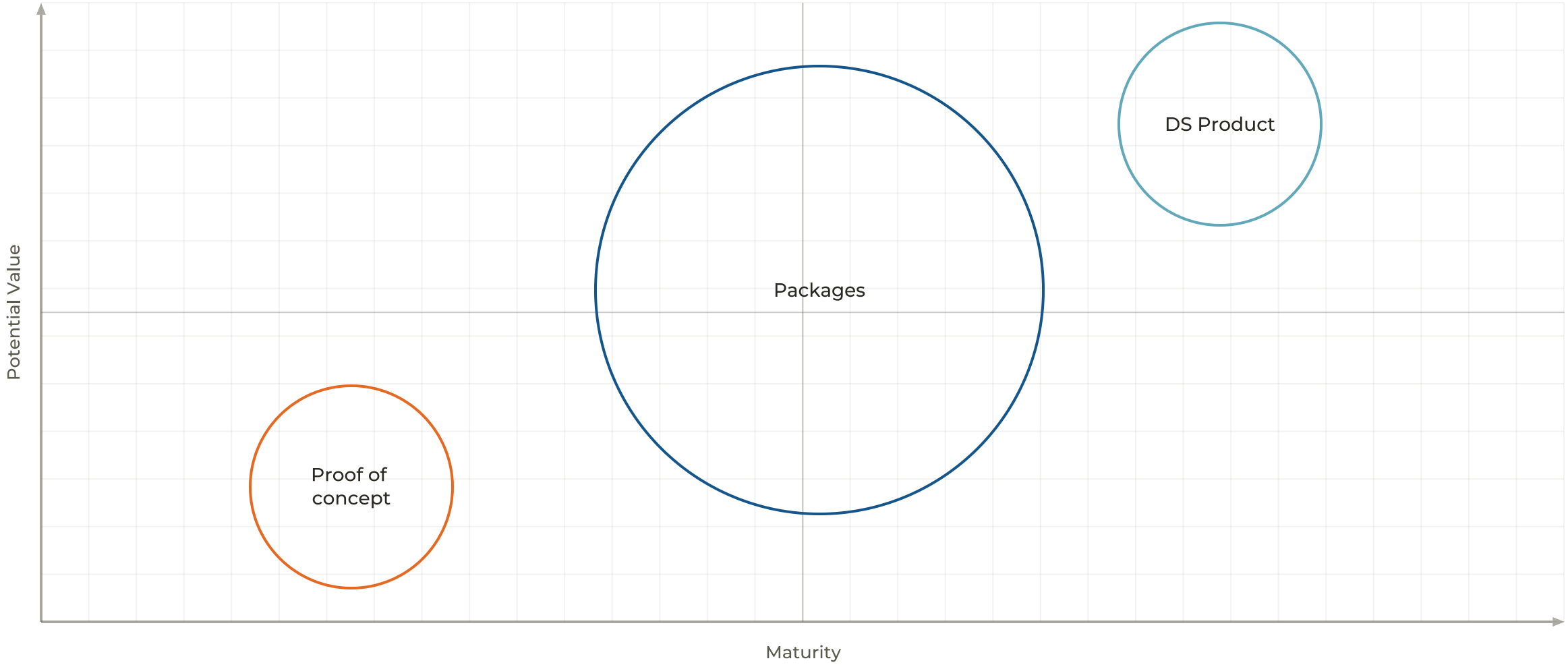
Raise awareness of the capability of an data science algorithm and pitch for increase adoption

Tasks	Nature of work
Data gathering	High uncertainty
Data Cleaning	Limited time & budget
Exploratory Data Analysis	Present to tell a story to small audience
Data Science Model	Limited automation & infrastructure
Metrics on Model Performance	Don't have to scale
	find use cases; estimate ROI

Project Went Well

- We need more of this
  - More data
  - More similar applications
- We need to automate this and make it available 24/7
- We need to let more people use this
  - How many?
  - How comfortable are they with programing?

From DS project to DS product



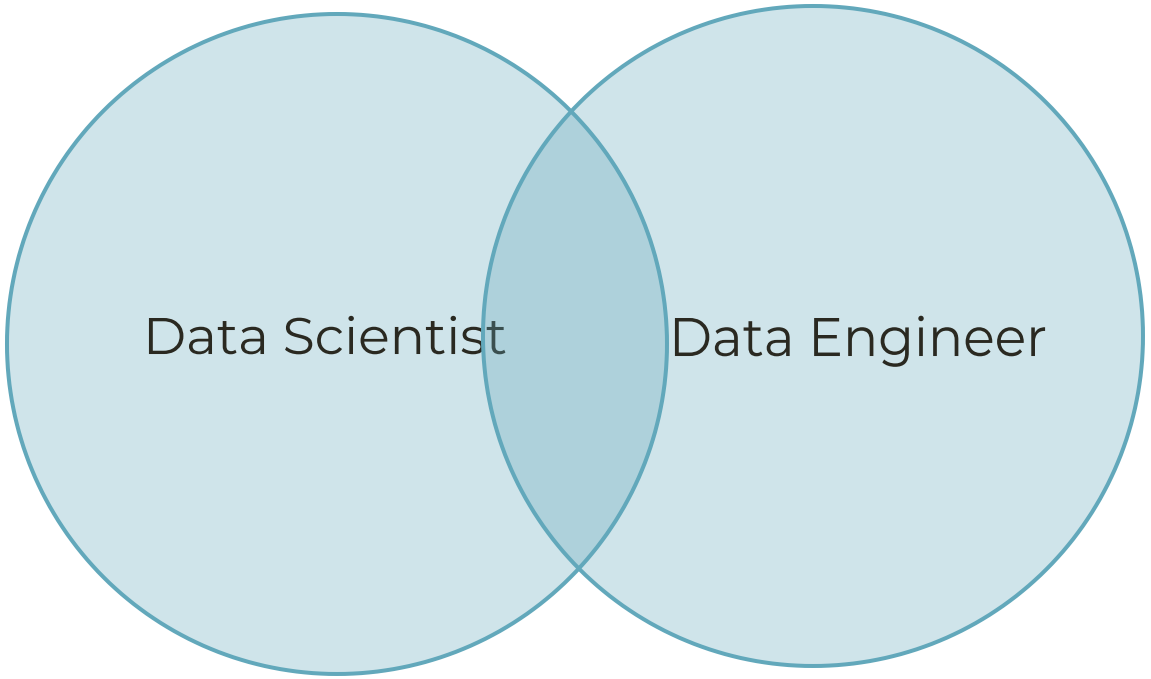
Scaling Data Science in Python

1. **Scaling with more data**
   work with spark, dask
2. **Generalize the function to a broader scope**
3. **Automate the data processing step on schedule or trigger**
   work with airflow
4. **Scale with more users**
   make packages discoverable, installable
   make no-code UI and dashboard
   make the function callable over http api
5. **Improve reliability and trust**
   testing

Data Project to Service

Input file	I/O Functions	Input Form
Data Transform Scripts	Data Transform Packages	Data Transform Pipeline
Train locally	Train in cluster	
Data Visualization Packages	Interactive Dashboard	
Save to Disk	Save to Database or Object Storage	

Who is responsible for package development



Section 1

Folder Structure of a Package

Scripts, modules, and packages

- **Scripts**

  A Python file that can be run in terminal with python scripts.py

- **Package**

  A directory full of Python code to be imported.

  E.g. Pandas

- **Subpackage**

  A smaller package inside a package

  E.g. pandas.testing

- **Module**

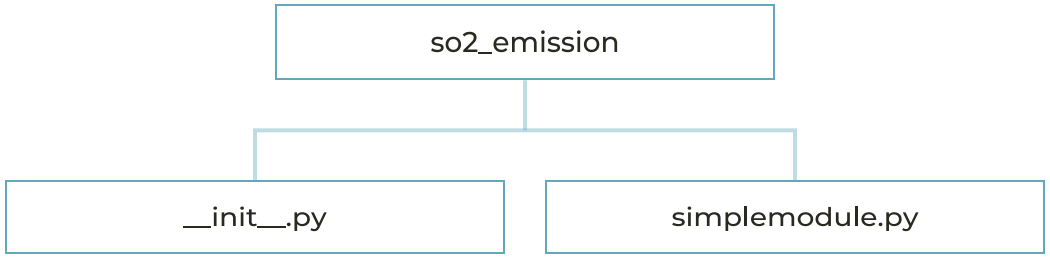
  A Python file which store the package code

- **Library**

  Either a package or a collection of packages

  E.g: Python standard library (math)

Directory tree of a package



__init__.py marks this directory as a Python package
simplemodule.py contains all the package code

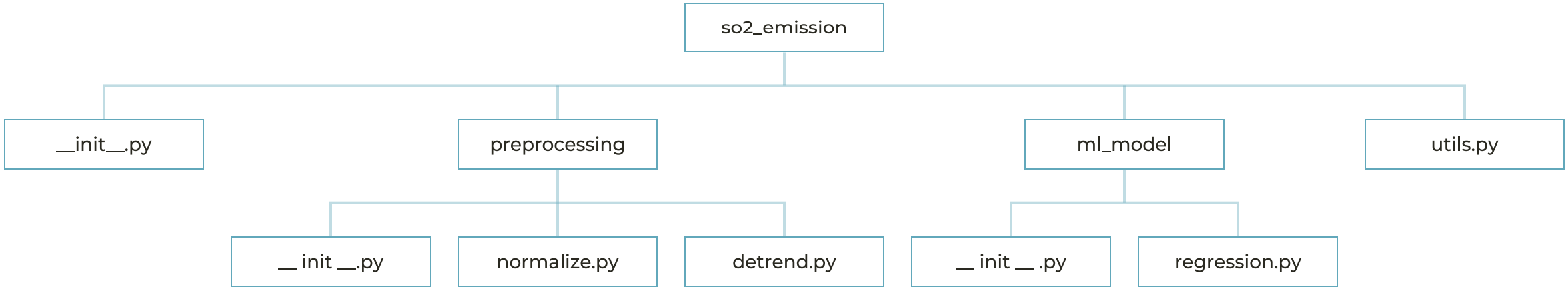
Contents of simple package

__ init __ .py

mymodule.py

```
def cool function():
...
return cool result
...
def another_cool_function():
...
return another_cool_result
```

Subpackages



__init__.py tells python this folder is a package
The absolute path inside the package starts from the parent package.
For example: my_package.preprocessing

Section 2

Documentation

Content

Why include documentation?

- Helps your users use your
  - Function
- What should be documented?
  - What does this function do?
  - What are the expected inputs?

```
import numpy as np
help(np.sum)
```

```
...
sum(a, axis=None, dtype=None, out=None)
Sum of array elements over a given axis.
Parameters
----------
a : array_like
    Elements to sum.
axis : None or int or tuple of ints, optional
    Axis or axes along which a sum is performed.
    The default, axis=None, will sum all of the
    elements of the input array.
...
```

Why include documentation?

- Helps your users use your
  - Class
- What should be documented?
  - What does this class do?
  - What are the expected inputs?
  - What are the expected outputs?
  - What errors can it raise?

```
import numpy as np
help(np.array)
```

```
...
array(object, dtype=None, copy=True)
Create an array.
Parameters
----------
object : array_like
    An array, any object exposing the array
    interface ...
dtype : data-type, optional
    The desired data-type for the array.
copy : bool, optional
    If true (default), then the object is copied
...
```

Why include documentation?

- Helps your users use your
  - Class method
- What should be documented?
  - What does this function do?
  - What are the expected inputs?
  - What are the expected outputs?
  - What errors can it raise?

```
import numpy as np
x = np.array([1,2,3,4])
help(x.mean)
```

```
...
mean(...) method of numpy.ndarray instance
a.mean(axis=None, dtype=None, out=None)
Returns the average of the array elements
along given axis.
Refer to `numpy.mean` for full documentation.
...
```

Function docstring

```
def count_words(filepath, words_list):
"""..."""
```

Function documentation

```
def count_words(filepath, words_list):
"""Count the total number of times these words appear."""
```

Function documentation

```
def count_words(filepath, words_list):
"""Count the total number of times these words appear.

The count is performed on a text file at the given location.
"""
```

Function documentation

```
def count_words(filepath, words_list):
"""Count the total number of times these words appear.

The count is performed on a text file at the given location.

[explain what filepath and words_list are]

[what is returned]
"""
```

Documentation Style

Numpy Style

```
"""Summary line.
Extended description of
function.
Parameters
----------
arg1 : int
Description of arg1 ...
```

Google Style

```
"""Summary line.
Extended description of
function.
Args:
arg1 (int): Description of
arg1
arg2 (str): Description of
arg2
```

reStructured text Style

```
"""Summary line.
Extended description
of function.
:param arg1:
Description of arg1
:type arg1: int
:param arg2:
Description of arg2
:type arg2: str
```

NumPy documentation style

Popular in Scientific Python Packages like

- numpy
- scipy
- pandas
- sklearn
- matplotlib
- dask

```
"""Summary line.
Extended description of
function.
Parameters
----------
arg1 : int
Description of arg1 ...
```

NumPy documentation style

```
import scipy
help(scipy.percentile)
```

```
percentile(a, q, axis=None, out=None, overwrite, input=False,
interpolation='linear')
Compute the q-th percentile of the data along the specified axis.
Returns the q-th percentile(s) of the array elements.
Parameters
----------
a : array_like
Input array or object that can be converted to an array
```

NumPy documentation style

```
import scipy
help(scipy.percentile)
```

```
percentile(a, q, axis=None, out=None, overwrite_input=False,
interpolation='linear')
...
Parameters
----------
...
axis : {int, tuple of int, None}
...
interpolation : {'linear','lower','higher','midpoint','nearest'}
```

List multiple types for parameter if appropriate

List accepted values if only a few valid options

NumPy documentation style

```
import scipy
help(scipy.percentile)
```

```
percentile(a, q, axis=None, out=None, overwrite_input=False,
interpolation='linear')
...
Returns
-------
percentile : scalar or ndarray
    If `q` is a single percentile and `axis=None`, then the result
    is a scalar. If multiple percentiles are given, first axis of
    the result corresponds to the percentiles...
...
```

NumPy documentation style

- Other Sections includes:
- Raises
- See Also
- Notes
- References
- Examples

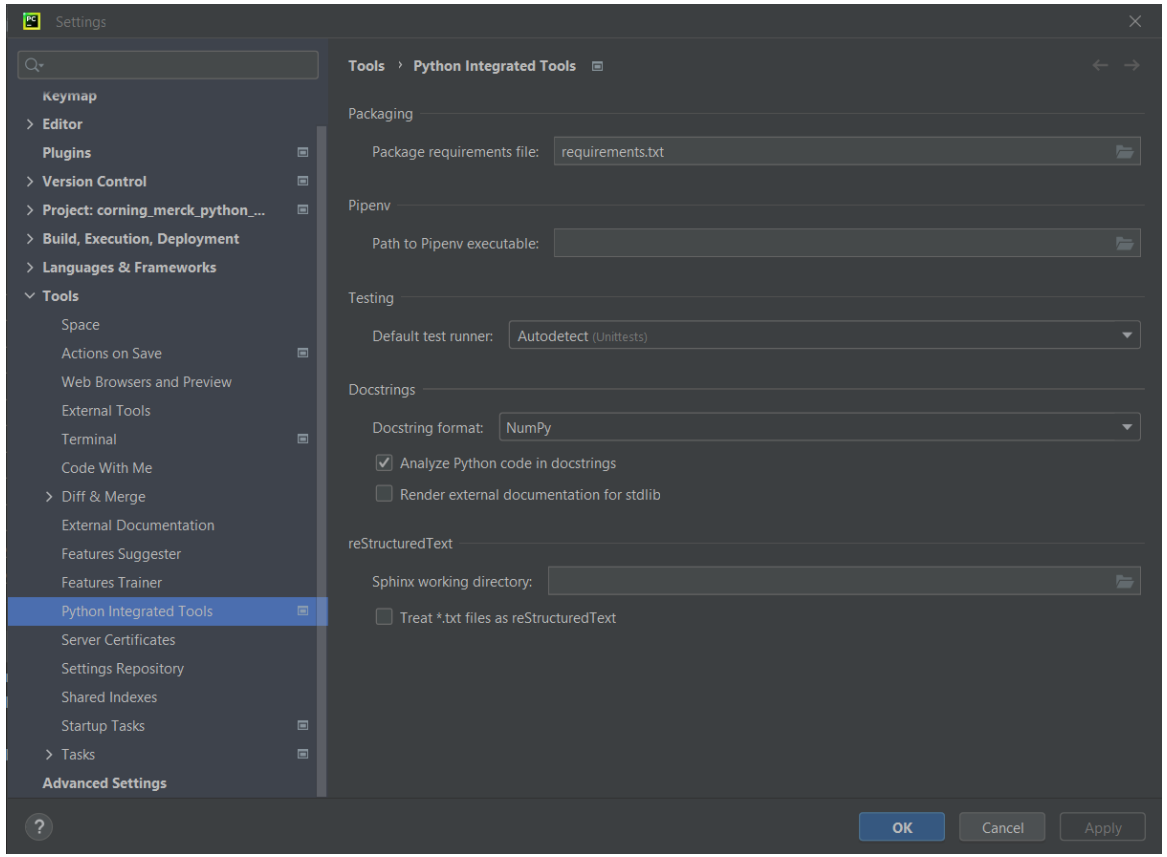
Example in docstring

```
def sum(list: list[float]) -> float:
    """Sum the value in a list
    Returns
    ....
    Examples
    -------------
    >>> a = [ 1.0, 2.0, 3.0]
    >>> sum(a):
    6.0

    """
```

Setting default documentation style in IDE

For example, in Pycharm File > Settings > Tools > Python Integrated Tools > Docstrings



Documentation templates and style translation

- **pyment** can be use to generate docstring
- Run from terminal
- Translate Any documentation style from one style to another

Type Hinting

```
def count_words(filepath: str, words_list: list) --> int:
    ...
```

- types hinting is optional
- Make code more readable
- IDE use type hint to support type checking, code completion and highlighting error
- Code quality tool like mypy perform type check in terminal or CI/CD
- common types are
  - str
  - list
  - int

    float

    dict

    bool

Type Hinting

```
from typing import List
```

```
WordList = List[str]

def count_words(file_paths: str, word_list: WordList) --> int:
    ...
```

```
def count_words(file_paths: str, word_list: List[str]) --> int:
    ...
```

Also works for Dict and Tuple

Type Hinting

```
from typing import Union
```

```
WordList = Union[str, float]

def count_words(file_paths: str, word_list: WordList) --> int:
    ...
```

Type Hinting

```
from typing import Optional
```

```
WordList = Optional[str]

def count_words(file_paths: str, word_list: Optional[str] = None) --> int:
    ...
```

- equivalent to Union[X, None]

Type Hinting

```
import numpy as np
```

```
def sum(array: np.array) --> int:
    ...
```

- Classes can be use as type
- Don't bother with complex classes like matplotlib objects
- Typing features changes between python versions.
  - See more: typing — Support for type hints — Python 3.10.4 documentation

Package, subpackage and module documentation

so2_emission/__init__.py

```
"""
Linear regression for
Python
==========================

my_package is a complete
package for implmenting
linear regression in
python.
```

so2_emission/preprocessing/__init__.py

```
"""
A subpackage for standard preprocessing
operations.
"""
```

so2_emission/preprocessing/normalize.py

```
"""
A module for normalizing data.
"""
```

Section 3

Structuring imports

Without package imports

```
import so2_emission
```

```
help(so2_emission.preprocessing)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: module 'so2_emission' has no
attribute 'preprocessing'
```

Directory tree for package with subpackages

```
so2_emission/
|--__init__.py
|-- preprocessing
| |--__init__.py
| |-- normalize.py
| |-- standardize.py
|-- regression
| |--__init__.py
| |-- regression.py
|-- utils.py
```

Without package imports

```
import so2_emission.preprocessing
```

```
help(so2_emission.preprocessing)
```

```
Help on package so2_emission.preprocessing in
my_package:
NAME
    so2_emission.preprocessing - A subpackage
for             standard preprocessing
operationss
...
```

Directory tree for package with subpackages

```
my_package/
|--__init__.py
|-- preprocessing
| |--__init__.py
| |-- normalize.py
| |-- standardize.py
|-- regression
| |--__init__.py
| |-- regression.py
|-- utils.py
```

Without package imports

```
import so2_emission.preprocessing
```

```
help(so2_emission.preprocessing.normalize)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: module
'so2_emission.preprocessing' has no attribute
'normalize'
```

Directory tree for package with subpackages

```
so2_emission/
|--__init__.py
|-- preprocessing
| |--__init__.py
| |-- normalize.py
| |-- standardize.py
|-- regression
| |--__init__.py
| |-- regression.py
|-- utils.py
```

Importing subpackages into packages

so2_emission/__init__ .py

Absolute import

```
from so2_emission import preprocessing
```

- Used most

Relative import

```
from . import preprocessing
```

Directory tree for package with subpackages

```
so2_emission/
|--__init__.py <--
|-- preprocessing
| |--__init__.py
| |-- normalize.py
| |-- standardize.py
|-- regression
| |--__init__.py
| |-- regression.py
|-- utils.py
```

Importing modules

We imported *preprocessing* into *so2_emission*, But *preprocessing* has no link to *normalize*

```
import so2_emission
help(so2_emission.preprocessing)
```

```
Help on package
so2_emission.preprocessing in
so2_emission:
NAME
so2_emission.preprocessing - A
subpackage
for standard preprocessing operations.
```

```
import so2_emission
help(so2_emission.preprocessing.normalize)
```

```
Traceback (most recent call last):
File "<stdin>"
, line 1, in <module>
AttributeError: module
'so2_emission.preprocessing' has no
attribute 'normalize'
```

Importing modules

so2_emission /preprocessing/__ init __ .py

Absolute import

```
from so2_emission.preprocessing import
normalize
```

Relative import

```
from . import normalize
```

Directory tree for package with subpackages

```
so2_emission/
|--__init__.py
|-- preprocessing
| |--__init__.py  <--
| |-- normalize.py
| |-- standardize.py
|-- regression
| |--__init__.py
| |-- regression.py
|-- utils.py
```

Restructuring imports

```
import so2_emission
help(so2_emission.preprocessing.normalize.normalize_data)
```

```
Help on function normalize_data in module
so2_emission.preprocessing.normalize:

normalize_data(x)

    Normalize the data array
```

Import function into subpackage

so2_emission /preprocessing/__ init __ .py

Absolute import

```
from so2_emission.preprocessing.normalize
import normalize_data
```

Relative import

```
from .normalize import normalize_data
```

Directory tree for package with subpackages

```
so2_emission/
|--__init__.py
|-- preprocessing
| |--__init__.py <--
| |-- normalize.py
| |-- standardize.py
|-- regression
| |--__init__.py
| |-- regression.py
|-- utils.py
```

Import function into subpackage

```
import so2_emission
help(so2_emission.preprocessing.normalize_data)
```

```
Help on function normalize_data in module
so2_emission_imp.preprocessing.normalize:

normalize_data(x)
  Normalize the data array
```

Importing between sibling modules

so2_emission/preprocessing/normalize.py

Absolute import

```
from so2_emission.preprocessing.funcs import
mymax, mymin
```

Relative import

```
from .funcs import mymax, mymin
```

Directory tree for package with subpackages

```
so2_emission/
|--__init__.py
|-- preprocessing
| |--__init__.py
| |-- normalize.py  <--
| |-- standardize.py
| |-- funcs.py
|-- regression
| |--__init__.py
| |-- regression.py
|-- utils.py
```

Importing between modules far apart

A custom exception MyException is in utils.py

In normalize.py, standardize.py and regression.py

Absolute import

```
from so2_emission.utils import MyException
```

Relative import

```
from ..utils import MyException
```

Directory tree for package with subpackages

```
so2_emission/
|--__init__.py
|-- preprocessing
| |--__init__.py
| |-- normalize.py  <--
| |-- standardize.py <--
|-- regression
| |--__init__.py
| |-- regression.py  <--
|-- utils.py
```

Section 4

Package

Package

1. Provide basic information
2. Dependency
3. Readme
4. Including non-python script in package

Why should you install your own package?

Inside example_script.py

```
import so2_emission
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named
'so2_emission'
```

Directory tree for package with subpackages

```
home/
|-- so2_emission <-- in same directory
| |--__init__.py
| |-- preprocessing
| | |--__init__.py
| | |-- normalize.py
| | |-- standardize.py
| |-- regression
| | |--__init__.py
| | |-- regression.py
| `-- utils.py
|-- example_script.py <-- in same
directory
```

setup.py

- Is used to install the package
- Contains metadata on the package

Package directory structure

Before

```
home/
|-- so2_emission
| |--__init__.py
| |-- preprocessing
| | |--__init__.py
| | |-- normalize.py
| | |-- standardize.py
| |-- regression
| | |--__init__.py
| | |-- regression.py
| `-- utils.py
|-- example_script.py <-- in same
directory
```

Add a outer folder layer and setup.py

```
home/
|--so2_emission    <-- outer directory
| |-- so2_emission <-- source code
directory
| | |--__init__.py
| | |-- preprocessing
| | | |--__init__.py
| | | |-- normalize.py
| | | |-- standardize.py
| | |-- regression
| | | |--__init__.py
| | | |-- regression.py
| | `-- utils.py
|  `-- setup.py    <-- setup script in
outer
|-- example_script.py
```

Inside setup.py

```
# Import required functions
from setuptools import setup
# Call setup function
setup(
author="Daniel Cho",
description="A package for so2 emission analysis.",
name="so2-emission",
version="0.1.0",
)
```

version number = (major number).(minor number).(patch number)

Where are the code?

```
# Import required functions
from setuptools import setup, find_packages
# Call setup function
setup(
author="Daniel Cho",
description="A complete package for linear regression.",
name="so2-emission",
version="0.1.0",
packages=find_packages(include=["so2_emission","so2_emission.*"]),
)
```

Editable installation

```
pip install -e .
```

- . means package in the current directory
- -e means editable.
  - changes in the package is immediately available to where the package is used on your computer
  - useful for developer when testing

```
home/
|--so2_emission    <-- navigate to here
| |-- so2_emission
| | |--__init__.py
| | |-- preprocessing
| | | |--__init__.py
| | | |-- normalize.py
| | | |-- standardize.py
| | |-- regression
| | | |--__init__.py
| | | |-- regression.py
| | `-- utils.py
|   `-- setup.py
|-- example_script.py
```

What are dependencies?

- Other packages you import inside your package
- Inside normalize.py

```
# These imported packages are dependencies
import numpy as np
import pandas as pd
...
```

Adding dependencies to setup.py

```
from setuptools import setup, find_packages
setup(
...
install_requires=['pandas','numpy'],
)
```

Controlling dependency version

```
from setuptools import setup, find_packages
setup(
...
install_requires=[
'pandas>=1.3',
'numpy==1.21',
'matplotlib>=3.5.1,<4'
)
```

Controlling dependency version

```
from setuptools import setup, find_packages
setup(
...
install_requires=[
'pandas>=1.4',          #good
'scipy==1.8',           #bad
'matplotlib>=3.5.1,<4' #good
)
```

- Allow as many package versions as possible
- Get rid of unused dependencies

Python versions

```
from setuptools import setup, find_packages
setup(
...
python_requires='>=3.7, !=3.10.*, !=3.9.*',
)
```

Choosing dependency and package versions

- Check the package history or release notes
- Test different versions
- Check package dev note future major releases

Making an environment for developers

Save package requirements to a file

```
pip freeze > requirements.txt
```

Install package requirements from a file

```
pip install -r
requirements.txt
```

```
home/
|--so2_emission    <-- navigate to here
| |-- so2_emission
| | |--__init__.py
| | |-- preprocessing
| | | |--__init__.py
| | | |-- normalize.py
| | | |-- standardize.py
| | |-- regression
| | | |--__init__.py
| | | |-- regression.py
| | `-- utils.py
| |-- setup.py
| `-- requirements.txt <-- developer
environment
|-- example_script.py
```

What is a README?

- The "front page" of your package
- Displayed on Gitlab or Github or PyPI
- Sections
  - Title
  - Description and Features
  - Installation
  - Usage examples
  - Contributing
  - License
- Helps people understands your package

Markdown Format

Content of README.md

````
# so2_emission
so2_emission is a package for complete
**linear regression** in Python.
You can find out more about this package
on [link](https://mylink.com)
## Installation
You can install this package using
```bash
pip install so2_emission
```
````

so2_emission

so2_emission is a package for complete **linear regression** in python.

You can find out more about this package on **link**

Installation

```
pip install so2_emission
```

MANIFEST.in

- Lists all the extra files to include in your package distribution
-

MANIFEST.in

Contents of MANIFEST.in

```
include LICENSE
include README.md
```

```
home/
|--so2_emission          <--
| |-- so2_emission
| | |--__init__.py
| | |-- preprocessing
| | | |--__init__.py
| | | |-- normalize.py
| | | |-- standardize.py
| | |-- regression
| | | |--__init__.py
| | | |-- regression.py
| | `-- utils.py
| |-- setup.py
| |-- README.md
| |-- LICENSE
| |-- MANIFEST.in
| `-- requirements.txt
|-- example_script.py
```

Publishing your package

PyPI

PyPI · The Python Package Index

- pip installs packages from here
- Anyone can upload packages
- You should upload your package as soon as it might be useful
- Index the name and description of setup.py

Distributions

Distribution package - a bundled version of your package which is ready to install.

Source distribution - a distribution package (.tar.gz) which is mostly your source code.

Wheel distribution - a distribution package (.whl) which has been processed to make it faster to install.

How to build distributions

```
python setup.py sdist bdist_wheel
```

- sdist = source distribution
- bdist_wheel = wheel distribution

```
so2_emission/
|-- so2_emission
|-- setup.py
|-- requirements.txt
|-- LICENSE
|-- README.md
|-- dist              <---
| |-- so2_emission-0.1.0-py3-none-any.whl
| |-- so2_emission-0.1.0.tar.gz
|-- build
| |-- so2_emission.egg-info
```

PyPI Package Registry

Upload your distributions to PyPI

```
pip install twine
twine upload dist/*
```

```
so2_emission/          <---
|-- so2_emission
|-- setup.py
|-- requirements.txt
|-- LICENSE
|-- README.md
|-- dist
| |-- so2_emission-0.1.0-py3-none-any.whl
| |-- so2_emission-0.1.0.tar.gz
|-- build
|-- so2_emission.egg-info
```

Gitlab PyPI Authentication

PyPI packages in the Package Registry | GitLab

Create a personal access token or other type of password

Create a file in ~/.pypirc or C:/Users/<your_username/.pypirc on Windows

```
[distutils]
index-servers =
    gitlab

[gitlab]
repository =
https://gitlab.example.com/api/v4/projects/<project_id>/packages/pypi

username = <your_personal_access_token_name>
password = <your_personal_access_token>

python3 -m twine upload --repository gitlab dist/*
```

How other people can install your package

Install Package from PyPI

```
pip install so2_emission
```

Install Package from Gitlab

```
pip install --index-url https://<personal_access_token_name>:
<personal_access_token>@gitlab.example.com/api/v4/projects/<project_id>/packages/pypi/simple
 --no-deps <package_name>
```

Install Package from .whl file

```
pip install
path/to/file.whl
```

Uninstall a package

```
pip uninstall <package name>
```

Benefit of building package

- enable dependency version control
- enable tool discovery
- reduce communication cost
- reduce multiple forks of the same tools

Section 4

Testing your package

Testing

1 | Why automate testing

2 | Create a simple test

3 | Read test report

4 | Assert statements for various situations

5 | Organize tests and run groups of them

6 | Setup and tear down with fixture

7 | Testing docstring example

How can we test an implementation?

```
def my_function(argument):
...
```

- Test manually on the interpreter
- Alternative - should write function to perform tests automatically

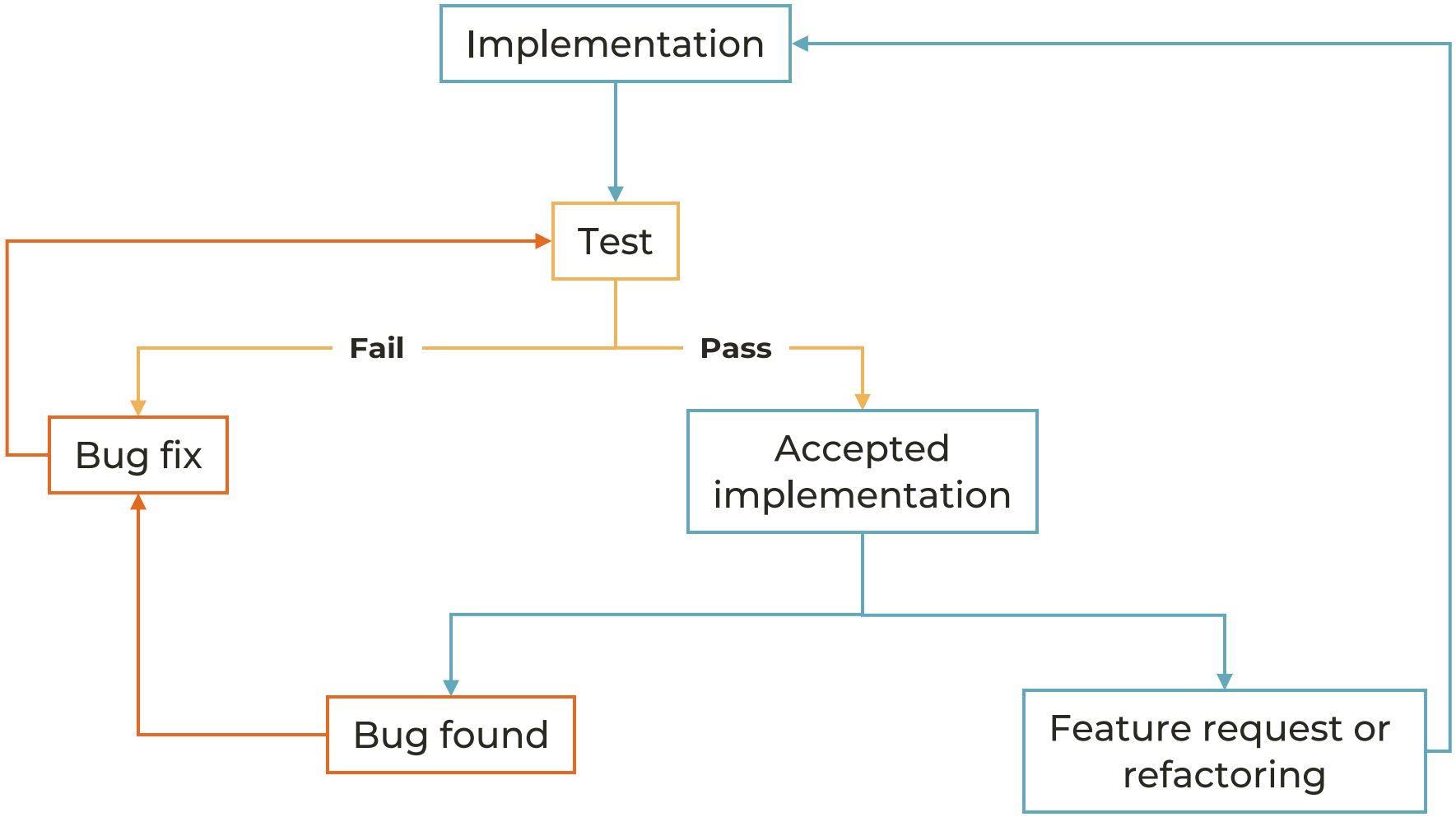
```
my_function(arg_1)
```

```
return_value_1
```

```
my_function(arg_2)
```

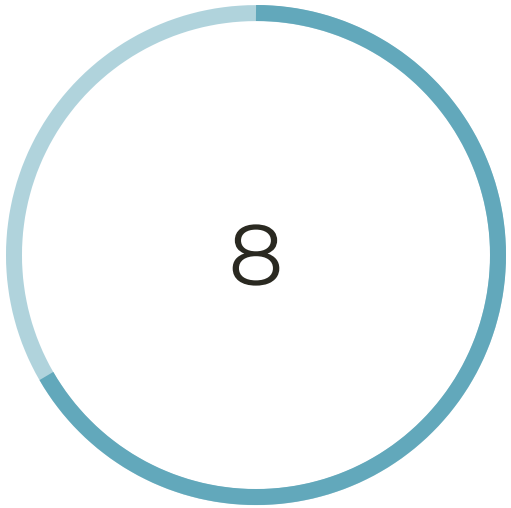
```
return_value_2
```

Life cycle of a function

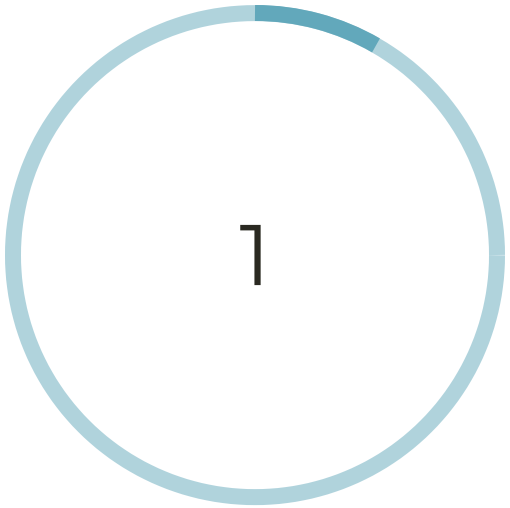


Automated tests save time and money

5 minute per test x 100

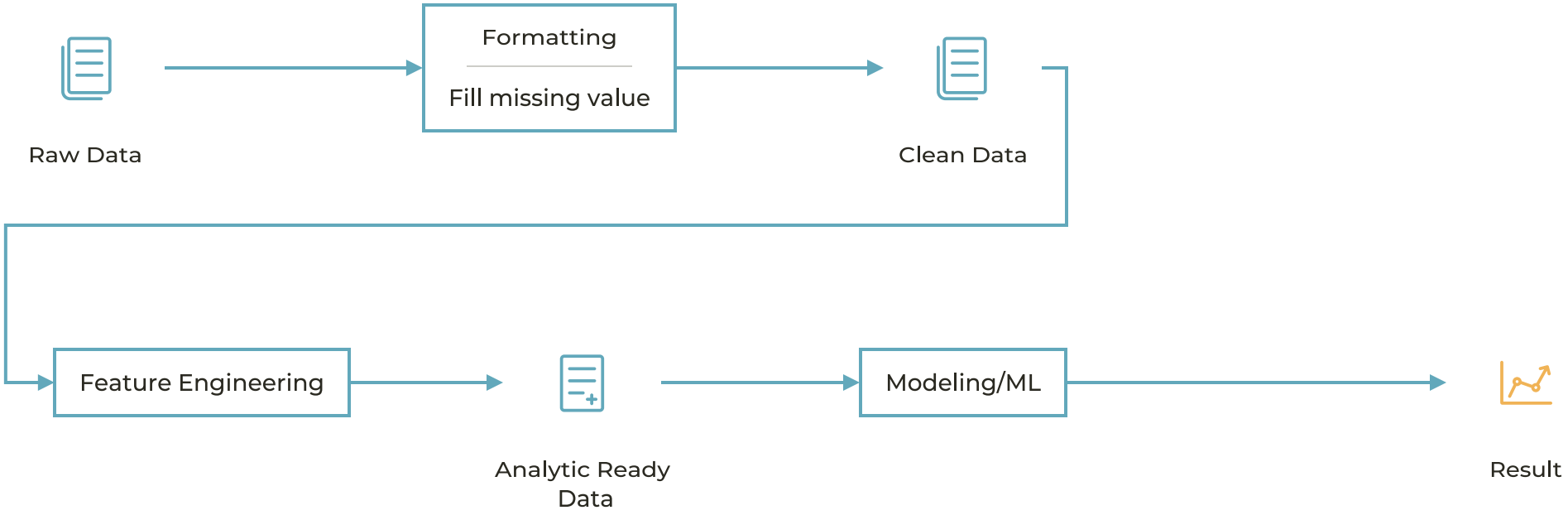


Manual Testing On The Interpreter

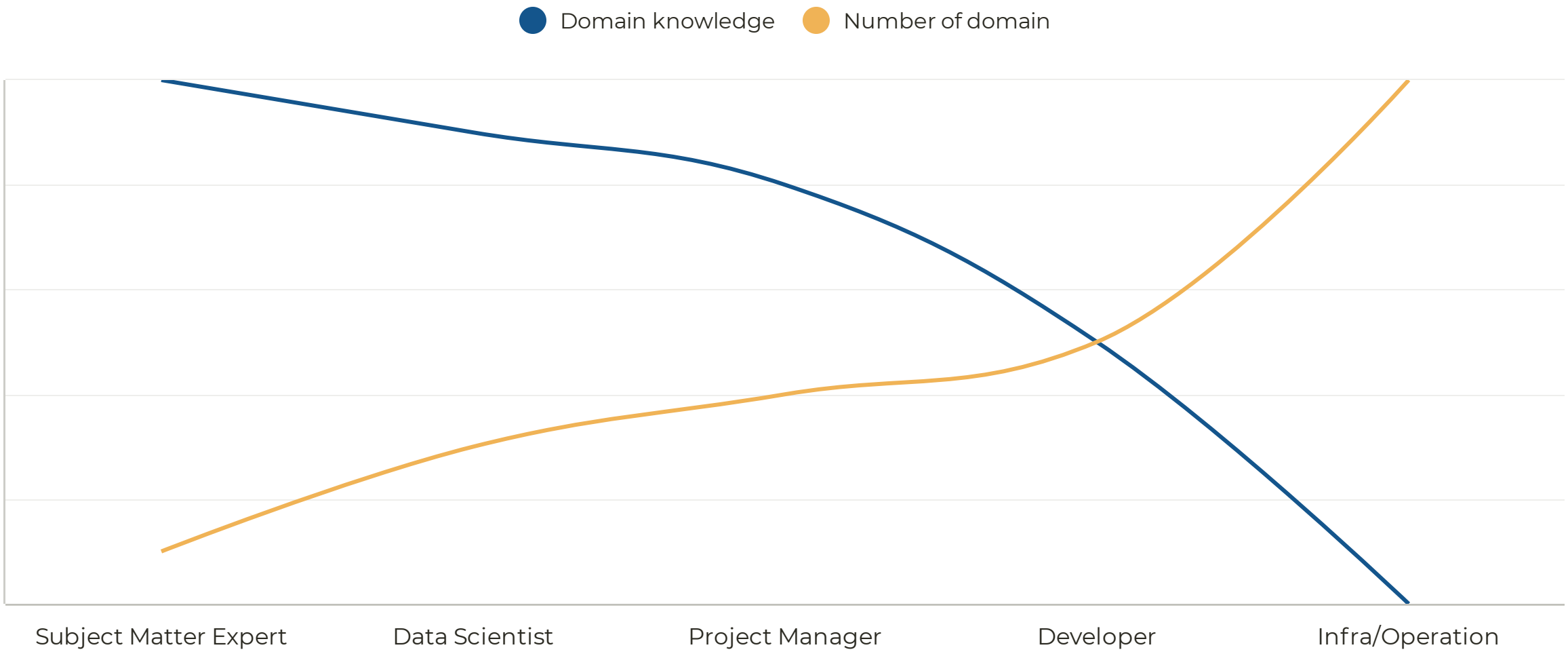


Automated Test

Pipeline Reliability



Who knows what to test?



Automated test as a communication tool

- Data Scientists communicate what and how to test
- Developers use tests to check pipeline implementation
- Toolchains engineers provide continuous integration testing environment
- Project managers estimate project maturity from the coverage and quality of tests
- New team members can use tests as example on how the api are used

Example

```
def row_to_list(row):
...
```

Argument	Type	Return value
42430\t2430\t857.32\n	Valid	[datetime_obj, 2430, 857.32]
42430.29167\t2406.4\t-999\n	Invalid	[datetime_obj, 2406.4, np.nan]

```
Time stamp      PumpFlow
PumpHead
42430   2430    857.32
42430.04167     2531.5  850.1
42430.08333     2579.7  845.75
42430.20833     2791.6  829.04
42430.25        2513.9  853.38
42430.29167     2406.4  -999
42430.33333     2583.6  846.47
42430.375       2642.2  840.79
42430.41667     2677.2  -999
42430.45833     2641.8  841.19
```

Testing on the console

```
row_to_list("42430\t2430\t857.32\n")
```

```
["3/1/2016  12:00:00 AM",2430,857.32]
```

```
row_to_list("42430.29167\t2406.4\t-999\n")
```

```
["3/1/2016  7:00:00 AM", 2406.4, np.nan]
```

Develop a complete test suite

Code directory layout

```
home/
so2_emission/
|-- __init__.py
|-- preprocessing/
|  |-- __init__.py
|  |-- func.py
|  |-- normalize.py
|  |-- standardize.py
```

Test directory layout

```
home/
tests/ # Test suite
|-- __init__.py
|-- preprocessing/
|  |-- __init__.py
|  |-- test_func.py
|  |-- test_normalize.py
|  |-- test_standardize.py
```

Mirror the test folder structure with source code

so2_emission and tests are inside the home folder

Python unit testing libraries

- pytest
- unittest
- doctest

We will use pytest!

- Has all essential features.
- Easiest to use
  - less verbose
- Most popular

Step 1: Create a file

- Create test test_func.py.
- "test_" indicate unit tests inside (naming convention).
- Also called test modules.

Step 2: Imports

Test module: test_func.py

```
import pytest
from so2_emission.preprocessing.func import row_is_valid
```

Step 3: Unit tests are Python functions

Test module: test_func.py

```
def test_row_is_valid():
```

Step 3: Unit tests are Python functions

Test module: test_func.py

```
def test_row_is_valid():
    row = "42430.29167\t2430\t857.32\n"
    assert row_is_valid(row) is True
```

Input	Type	Return
42430\t2430\t857.32\n	Valid	[datetime_obj, 2430, 857.32]

Assertion Returns

```
assert True
```

```
Type code
```

```
assert False
```

```
Traceback (most recent call last):
File "<stdin>"
, line 1, in <module>
AssertionError
```

Step 4: Unit tests are Python functions

Test module: test_func.py

```
def test_row_is_valid_false_case():
    row_is_false = "42430.29167\t2406.4\t-
999\n"
    assert row_is_valid(row_is_false) is False
```

Input	Type	Return
42430\t2430\t857.32\n	Valid	[datetime_obj, 2430, 857.32]
42430.29167\t2406.4\t-999\n	Invalid	[datetime_obj, 2406.4, np.nan]

Checking for None values

Do this to check if variable is none

```
assert var is None
```

Do not do this

```
assert var == None
```

Step 5: Running unit tests

Navigate to the folder containing the test, in terminal

```
pytest test_func.py
```

Test directory layout

```
home/
...
tests/ # Test suite
|-- __init__.py
|-- preprocessing/   <----
|  |-- __init__.py
|  |-- test_func.py
|  |-- test_normalize.py
|  |-- test_standardize.py
```

Understanding test result report

Test result report

```
(environment) PS
C:\Users\choy3\PycharmProjects\corning_merck_python_workshop\module4_testing\test\preprocessing>
 pytest test_func.py
==================================== test session starts
====================================
platform win32 -- Python 3.7.12, pytest-7.1.1, pluggy-1.0.0
rootdir: C:\Users\choy3\PycharmProjects\corning_merck_python_workshop\module4_testing
collected 2 items
test_func.py FF
[100%]
========================================= FAILURES
=========================================
_____________________________________ test_row_is_valid
_____________________________________

    def test_row_is_valid():
        row = "42430.29167\t2430\t857.32\n"
>       assert row_is_valid(row) is True
E       AssertionError: assert None is True
E        +  where None = row_is_valid('42430.29167\t2430\t857.32\n')
```

Section 1: general information

```
==================================== test session starts
====================================
platform win32 -- Python 3.7.12, pytest-7.1.1, pluggy-1.0.0
rootdir: C:\Users\choy3\PycharmProjects\corning_merck_python_workshop\module4_testing
collected 2 items
test_func.py FF
[100%]
```

Character	Meaning	When	Action
F	Failure	An exception is raised when running unit test.	Fix the function or the unit test
.	Passed	No exception raised when running unit test	No action required

Section 2: Information on failed tests

```
=================================== FAILURES
=========================================
_____________________________________ test_row_is_valid
_____________________________________

    def test_row_is_valid():
        row = "42430.29167\t2430\t857.32\n"
>       assert row_is_valid(row) is True
E       AssertionError: assert None is True
E        +  where None = row_is_valid('42430.29167\t2430\t857.32\n')

test_func.py:9: AssertionError
```

- The line raising the exception is marked by **>** .

- The error is that row_is_valid returned None when True is expected

  test_func.py:**9** means the error is located at line 9

Section 3: Summary of test result

```
=================================== short test summary info
================================
FAILED test_func.py::test_row_is_valid - AssertionError: assert None is True
FAILED test_func.py::test_row_is_valid_false_case - AssertionError: assert None is
False
=================================== 2 failed in 1.89s
====================================
```

- A short summary of which tests were run and their status and error
- Total number of test run and how long it took: 1.89s
  - Much faster than testing on the interpreter when running large number of tests

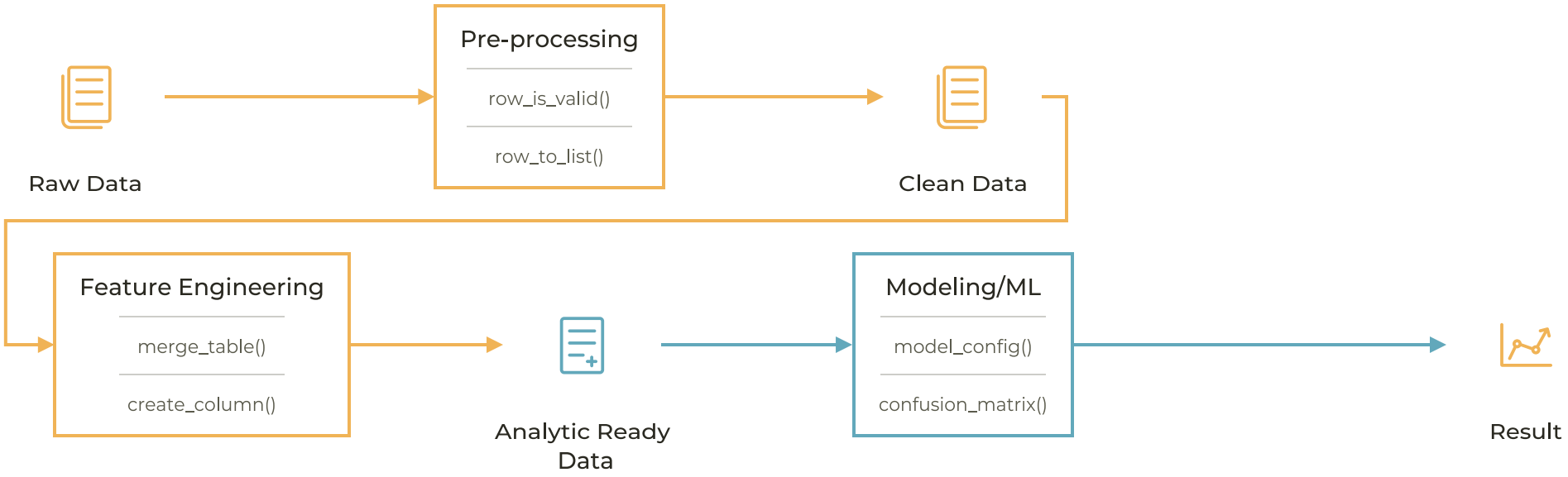
Automatically test code

Reduce down time
Increase reliability and trust

What is a unit?

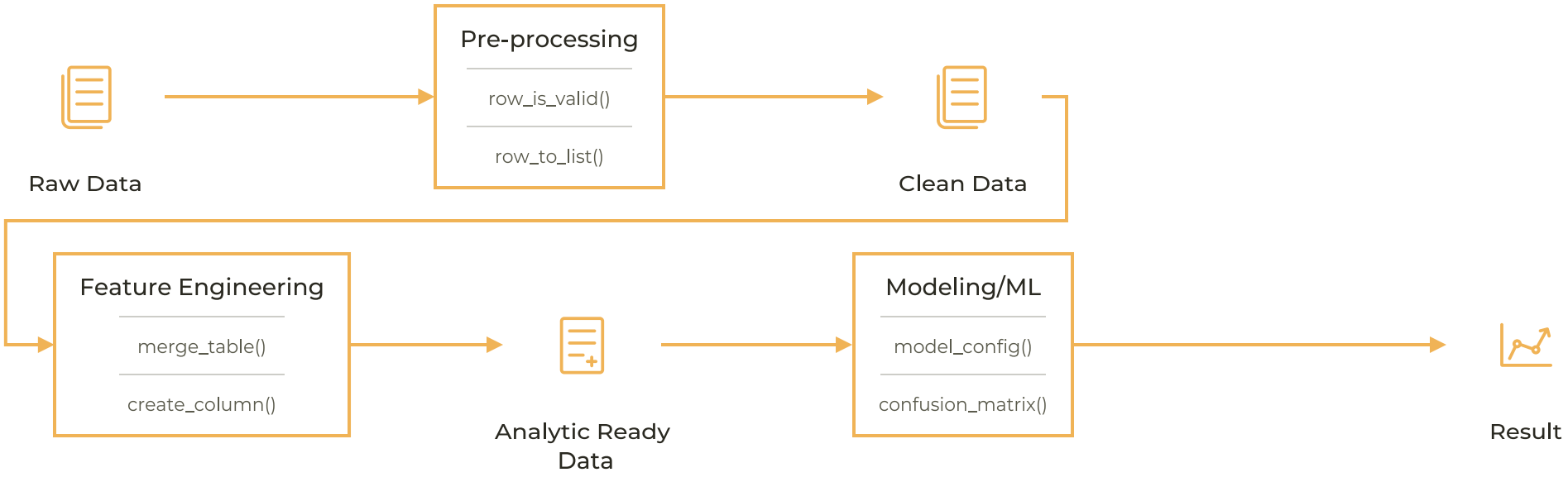
- Small, independent piece of code.
- Python function or class.

Integration Test



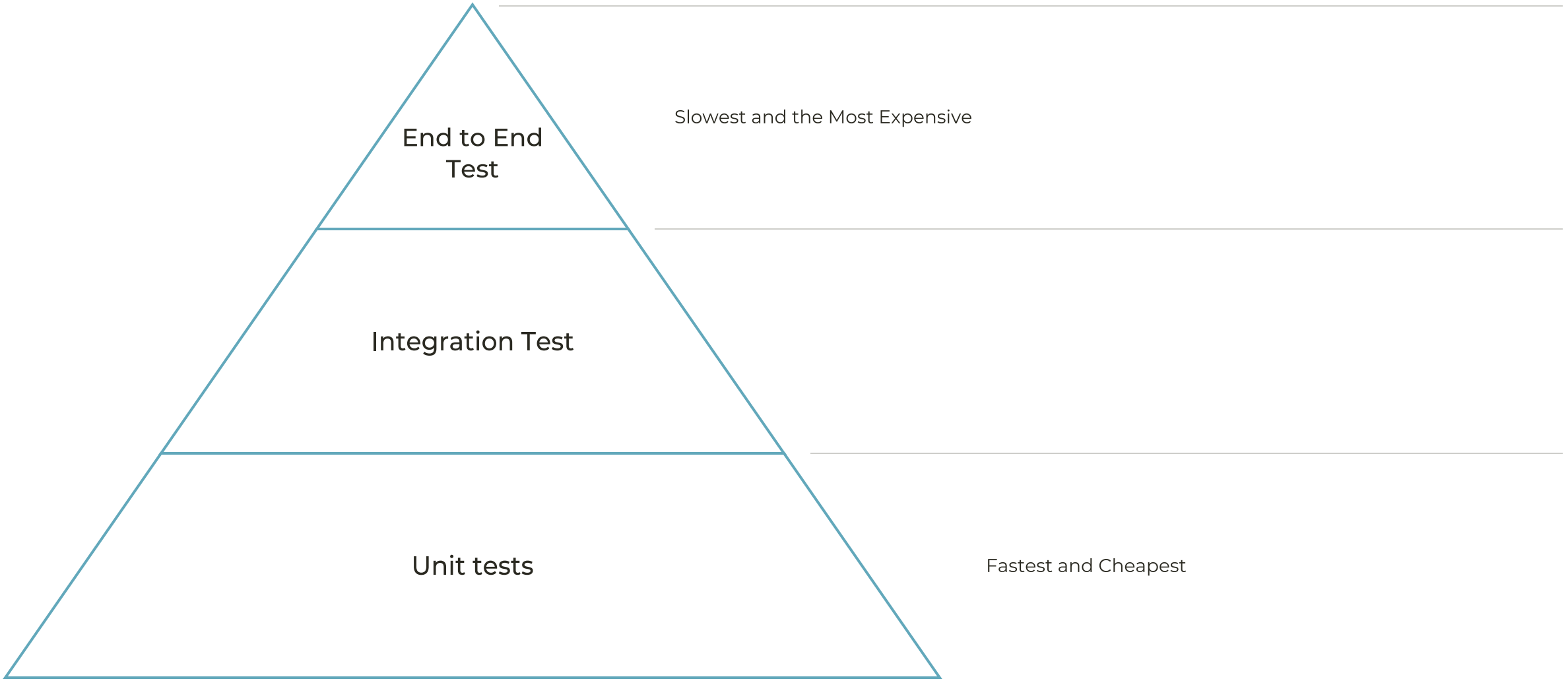
Tests if components work with each other

End-To-End Test



The exact classification may varies from team to team

Ideal Proportion of Test



Breaks down functions into helper functions

Original

```
def analysis(path: str):
    df = pd.read_csv(path)
    df['col_1'] = df['col_2'] + df['col_3']
    df.loc[condition1,'col4'] = ....
    total = df['col_1'].sum()
    plot = df.plot(...)
    return plot

def test_analysis():
    path = 'path/to/file'
    plot = analysis(path)
    assert plot?...
```

Test the transform in memory without read/write files

With sub functions

```
def _transform(df:pd.Dataframe):
    df['col_1'] = df['col_2'] + df['col_3']
    df.loc[condition1,'col4'] = ....
    return df

def analysis(path: str):
    df = pd.read_csv(path)
    df = _transform(df)
    total = df.agg(func=func1)
    plot = df.plot(...)
    return plot

def test_transform():
    test_df = pd.DataFrame(...)
    result = _transform(test_df)
    assert ....
```

Separate decision making from execution

Original

```
import sqlalchemy as sa
def change_state(input, connection: str):
    """issue different sql statement based
on complex input logic"""
    engine = create_engine()
    ..
    if cond1:
       sa.add()
    elif cond2:
       sa.add()
    session.commit()
def test_change_state():
    """ ensure logic is right"""
    connection = usr@pw.
    change_state(path)
    result = query_state_from_db
    assert result ...
```

With sub functions

```
def _logic(input):
    if cond1:
       ...
    elif cond2:
       ...
    return decisions
def change_state(input, connection: str):
    engine = create_engine()
    decisions = _logic(input)
    execute(decisions)
    return

def test_logic():
    """ ensure logic is right"""
    input1 = "..."
    decisions = _logic(input1)
    assert decisions is True
```

Unit Test Priority

- Start testing early in your project
- Test the code we or our group developed
- Test implementation of core business logics
- Test a subset of the data that represents most failure mode
- Start with one test per function
- It is not uncommon for established package to have a unit test suite larger than the actual package

Mastering Assert Statements

Structure of an assertion statement

```
assert boolean_expression
```

optional message argument

```
assert boolean_expression, message
```

```
assert 1 == 2,"One is not equal to two!"
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AssertionError: One is not equal to two!
```

```
assert 1 == 1,"This will not be printed since assertion passes"
```

```
Type code
```

Adding message to assert statement

Test module: test_func.py

Original

```
import pytest
...
def test_row_is_valid_false_case():
    row_is_false = "42430.29167\t2406.4\t-
999\n"
    assert row_is_valid(row_is_false) is
False
```

With message

```
import pytest
...
def test_row_is_valid_false_case():
    row_is_false = "42430.29167\t2406.4\t-
999\n"
    actual = row_is_valid(row_is_false)
    expected = False
    message = "-999 is an error code.
Result should be false"
    assert  actual is expected, message
```

Recommendations

- Include a message with assert statements.
- Print values of any variable that is relevant to debugging.

Beware of float return values!

```
0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

floating point error

```
0.1 + 0.1 + 0.1 == 0.3
```

```
0.30000000000000004
```

Do this instead

use pytest.approx() to wrap expected return value

```
assert 0.1 + 0.1 + 0.1 == pytest.approx(0.3)
```

For Numpy Array

```
assert np.array([0.1 + 0.1, 0.1 + 0.1 + 0.1]) == pytest.approx(np.array([0.2, 0.3]))
```

Numpy. testing framework

Test Support (numpy.testing) — NumPy v1.22 Manual

```
import numpy as np
...
np.testing.assert_allclose(np.array([0.1 + 0.1, 0.1 + 0.1 + 0.1]),np.array([0.2, 0.3]),
rtol=1e-07, atol=0)
```

- compares the two object to the accuracy of atol + rtol * abs(desired_value)

Testing Pandas Dataframe

Pandas.testing framework

- can compare float by default
- can compare values even if they are different data type

```
import pandas as pd
from pandas.testing import assert_frame_equal
df1 = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})
df2 = pd.DataFrame({'a': [1, 2], 'b': [3.0, 4.0]})
pandas.testing.assert_frame_equal(df1, df2)
```

Multiple assertions in one unit test

```
def test_row_to_list():
    row = "42430.29167\t2430\t857.32\n"
    row_to_list_result = row_to_list(row)

    assert isinstance(row_to_list_result, list)
    assert isinstance(row_to_list_result[0], datetime)
    expected_dt = datetime(year=2016, month=3, day=1, hour=0)
    assert row_to_list_result[0] == expected_dt
    ...
```

- Test will pass only if all assertions pass.
- Stop at the first failure

Testing for exceptions instead of return values

Testing for exceptions

```
import numpy as np
example_argument = np.array([2081, 314942, 1059, 186606, 1148, 206186]) # one dimensional
split_into_training_and_testing_sets(example_argument)
```

- Function split_into_training_and_testing_sets expects a two dimentions array
- expects to raise a ValueError

```
ValueError: Argument data array must be two dimensional. Got 1 dimensional array instead!
```

- let's check if it happens in pytest
  - uses a with statement/ context manager

with statement / context manager

```
with____:
    print("This is part of the context") # any code inside is the context
```

with statement / context manager

```
with context_manager:
    # <--- Runs code on entering context
    print("This is part of the context") # any code inside is the context
    # <--- Runs code on exiting context
```

Pytest.raises context manager

```
with pytest.raises(ValueError):
    # <--- Does nothing on entering the context
    "Test code goes here"
    # <--- If context raised ValueError, silence it.
    # <--- If the context did not raise ValueError, raise an exception.
```

Pytest.raises context manager

```
def test_valueerror_on_one_dimensional_argument():
    example_argument = np.array([2081, 314942, 1059, 186606, 1148, 206186])
    with pytest.raises(ValueError):
        split_into_training_and_testing_sets(example_argument)
```

- If function raises expected ValueError , test will pass.
- If function is buggy and does not raise ValueError , test will fail.

Test return length

```
def test_return_length():
    example_argument_value = np.array([[2081, 314942],[1059, 186606],[1148, 206186],])
    train, test = split_into_training_and_testing_sets(example_argument_value)
    assert len(train) == 2
    assert len(test) == 1
```

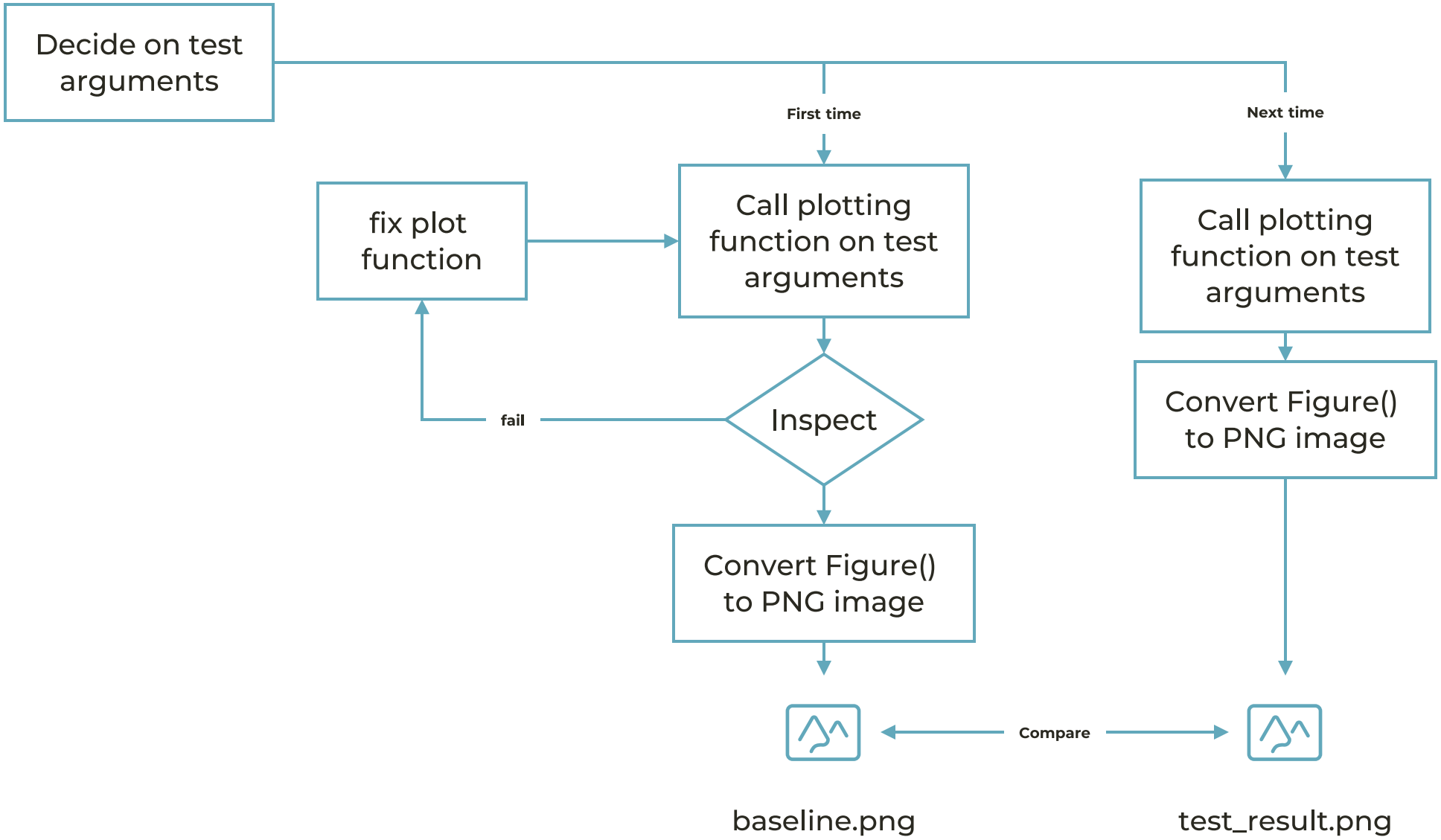
Testing by Comparing Matplotlib Images

pytest-mpl

- The reference image is subtracted from the generated image,
- The RMS of the residual is compared to a user-specified tolerance.

```
pip install pytest-mpl
```

Testing by comparing plot automatically



An example test

```python
import pytest
import numpy as np
from visualization import get_plot_for_best_fit_line

@pytest.mark.mpl_image_compare # Under the hood baseline generation and comparison
def test_plot_for_linear_data():
    slope = 2.0
    intercept = 1.0
    x_array = np.array([1.0, 2.0, 3.0]) # Linear data set
    y_array = np.array([3.0, 5.0, 7.0])
    title = "Test plot for linear data"
    return get_plot_for_best_fit_line(slope, intercept, x_array, y_array, title)
```

Generating the baseline image

In terminal

```
pytest -k "test_plot_for_linear_data"
--mpl-generate-path
visualization/baseline
```

```
home/
so2_emission/
tests/
|-- pre_processing/
|-- ml_models/
|-- visualization
|   |--__init__.py
|   |-- test_plots.py # Test module
|   |-- baseline # Contains baselines
|        |--
test_plot_for_linear_data.png
```

Run the test

```
pytest -k "test_plot_for_linear_data" --mpl
```

```
======================== test session starts ========================
...
collected 2 items / 1 deselected / 1 selected
visualization/test_plots.py . [100%]
============= 1 passed, 1 deselected in 0.68 seconds =============
```

Reading failure reports

```
pytest path-to-test-module --mpl
```

```
============================ FAILURES ============================
_______ TestGetPlotForBestFitLine.test_plot_for_linear_data _______
Error: Image files did not match.
RMS Value: 11.191347848524174
Expected:
/tmp/tmplcbtsb10/baseline-test_plot_for_linear_data.png
Actual:
/tmp/tmplcbtsb10/test_plot_for_linear_data.png
Difference:
/tmp/tmplcbtsb10/test_plot_for_linear_data-failed-diff.png
Tolerance:
2
============= 1 failed, 2 deselected in 1.13 seconds =============
```

A well tested function

Argument types

- Normal argument
- Bad argument
  - raise an error
- Special argument : implement special cases or logics

Test argument table

arguments	Type	Output(s)	Execption
...	bad	...	Value error
...	normal	...	--

Test-Driven Development

Problem

- Testing keep getting postpone because dev focus on creating features

Solution

- Create test argument table and test modules before creating the actual function
- Run the test and see it fails
- Develop the function until it passes all tests

For DS or ML project

- can be used for deterministic components
- model development is iterative

Organize tests into suite

Flat Test module

- Store all tests as function
- use the original function name that it tests

```
import pytest
from so2_emission.preprocessing.func import
row_is_valid, row_to_list
from datetime import datetime

def test_row_is_valid():
    row = "42430.29167\t2430\t857.32\n"
    assert row_is_valid(row) is True

def test_row_is_valid_false_case():
    row_is_false = "42430.29167\t2406.4\t-999\n"
    assert row_is_valid(row_is_false) is False

def test_row_to_list():
    row = "42430.29167\t2430\t857.32\n"
    row_to_list_result = row_to_list(row)

    assert isinstance(row_to_list_result, list)
    assert isinstance(row_to_list_result[0],
datetime)
```

Test Class

Use Class to store a number of tests for the same function

Test Class is like a folder

Test Class

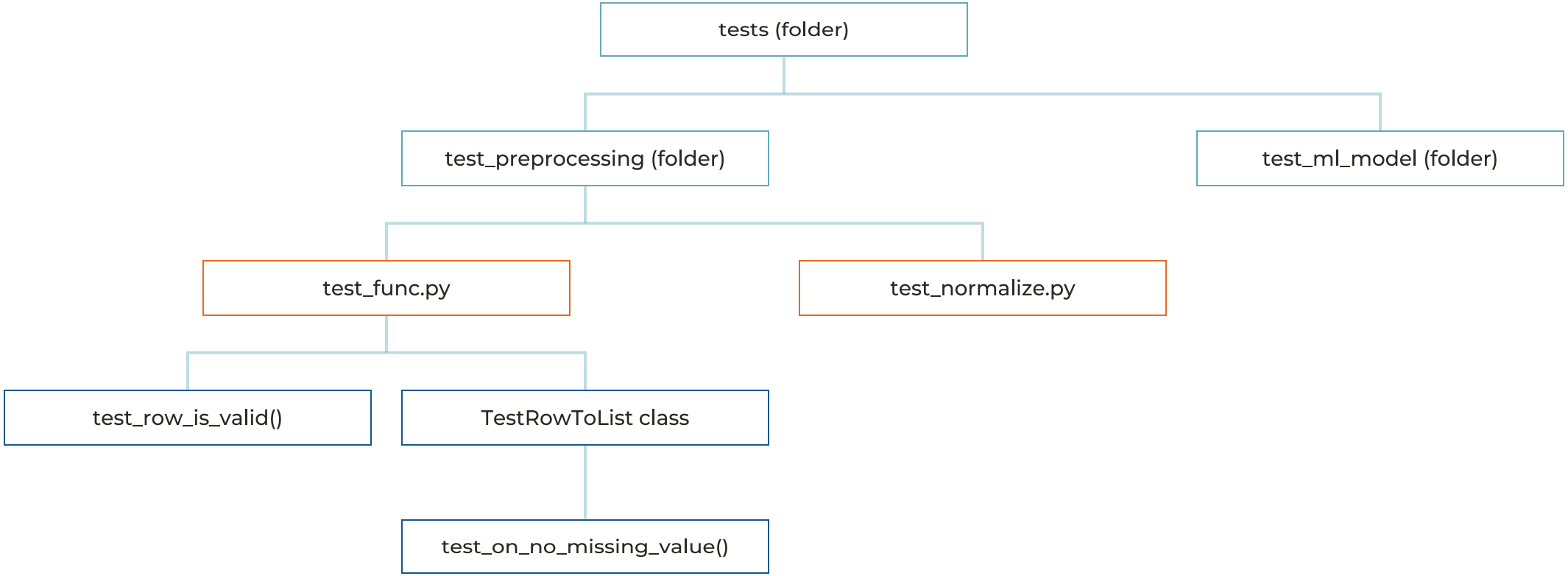
```
import pytest
from so2_emission.preprocessing.func import row_is_valid, row_to_list
from datetime import datetime

class TestRowToList(object): # Use CamelCase, always put the argument object
    def test_on_no_tab_no_missing_value(self): # A test for row_to_list(), always use argument
self
    ...
    def test_on_two_tabs_no_missing_value(self): # Another test for row_to_list(), argument is
self
    ...

class TestAnotherFunction(object):
    def test_case_one(self):
    ...
```

Create another class for another function

Test execution



Running all tests

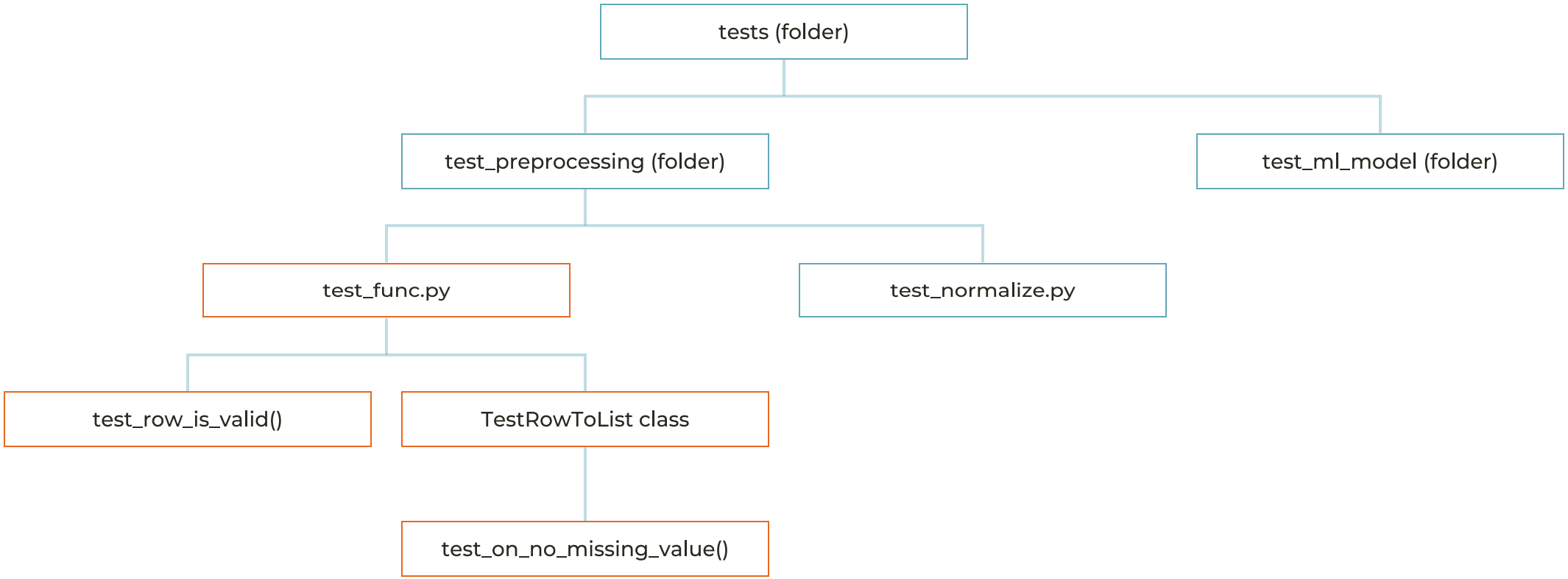
```
cd tests
pytest
```

- Recurses into directory subtree of tests/ .
- Filenames starting with test_ → test module.
- Classnames starting with Test → test class.
- Function names starting with test_ → unit test.

The -x flag: stop after first failure

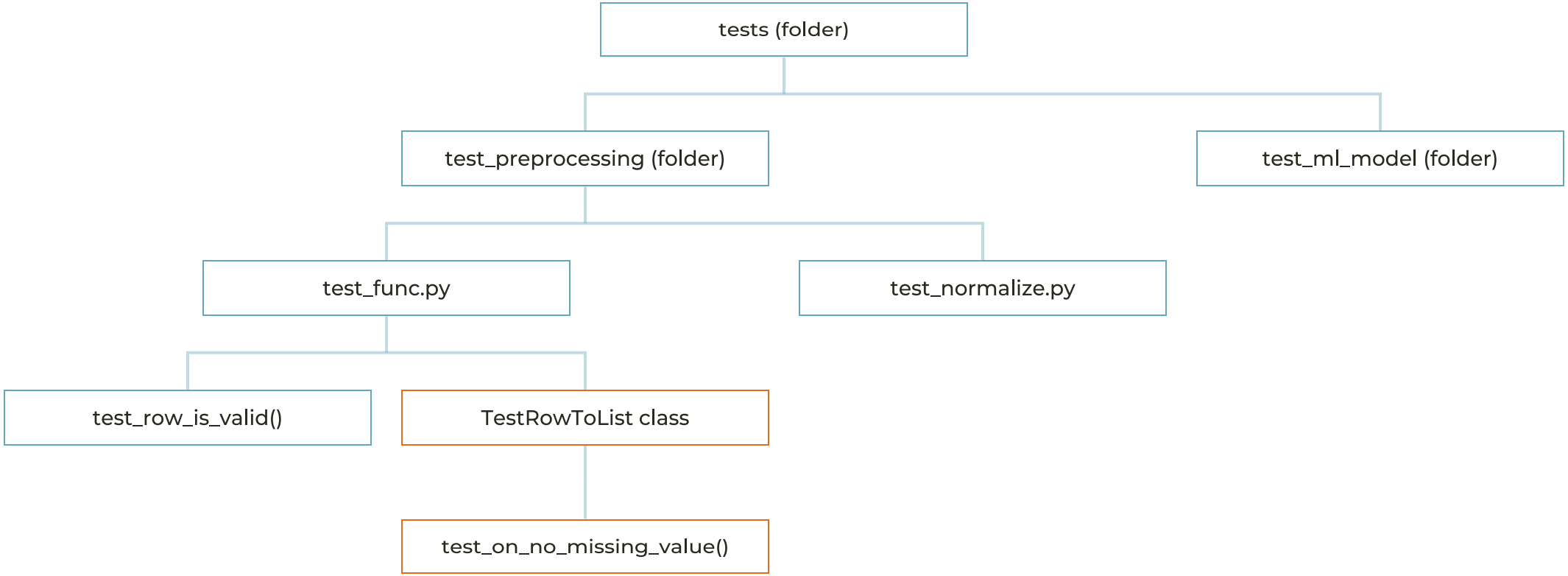
```
pytest -x
```

Runing a test module



pytest tests/test_preprocessing/test_func.py

Runing a test module or unit test

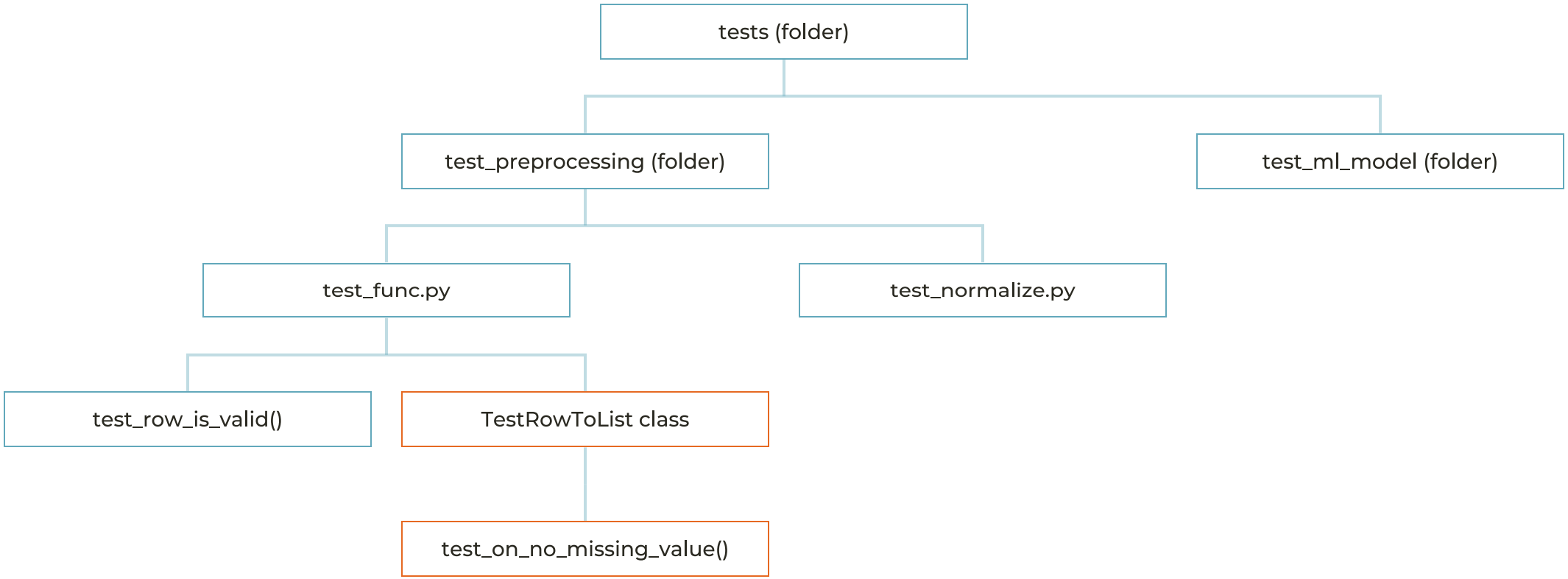


Node ID

Node ID for a unit test : <path to test module>::<test class name>::<unit test name>

Node ID for a test class: <path to test module>::<test class name>

Running a unit test




```
pytest tests/test_preprocessing/test_func.py::TestRowToList::test_on_no_missing_val
```

Run using keyword expression -k

```
pytest -k "pattern"
```

- Runs all tests whose node ID matches the pattern
- case insensitive
- can't limit search to function name or class name
  - can't differentiate Test_Row_To_List class and test_row_to_list function
  - Use TestClass::test_method naming convention
- Example:
  - pytest -k "test_on_no_missing_value"
- Accept substrings
  - pytest -k "test_on_no_missing"
- Accept "and", "not" operators

    pytest -k "TestRowToList and not test_on_no_missing"

    runs all test in TestRowToList class except test_on_no_missing_value()

Run using keyword expression -k

```
pytest -k "pattern"
```

- Runs all tests whose node ID matches the pattern
- case insensitive
- can't limit search to function name or class name
  - can't differentiate Test_Row_To_List class and test_row_to_list function
  - Use TestClass::test_method naming convention
- test names should be unique
- can find substring

Other useful pytest decorators

- expected to fail
  - @pytest.mark.xfail
  - Test will be marked as x instead of F in summary statistics
  - Test suite stay green
- skip test conditionally
  - @pytest.mark.skipif(condition bool)
  - Test will be skipped if condition is true
  - Use for testing that only runs on certain python version, OS
- Can provide reason for both decorators

```
@pytest.mark.___ <-- applies to class
class TestRowToList(object)
    @pytest.mark.___ <-- applies to a
test
    def test_on_no_missing_value()
        ...
```

Tests environment setup and tear down

Testing that read or write to file system

Possible use case

- testing file paths
- testing end-to-end feature
- testing functions that create folder structures

Preparing the environment

- Create a temp folder
- Add the expected folder structure and files
- known as setup
- Run test in the temp folder

Clean up after test

- Remove everything after test even if test fails
  known as teardown

The environment

pytest fixture

Create a fixture function in your test module

```
import pytest

@pytest.fixture
def my_fixture():
    # Do setup here
    yield data # Use yield instead of
return
    # Do teardown here


def test_something(my_fixture):
...
data = my_fixture
...
```

pytest fixture

- A fixture can call another fixture
- Need to invoke fixture in the test

- pytest execution sequence
  - temp_folder --> add_files -- > test_something --> tear down

```
import pytest

@pytest.fixture
def temp_folder():
# create a temp folder
  yield path

@pytest.fixture
def my_files(temp_folder):
path = temp_folder / "file.txt"
shutil.copy(source,path)
yield path # Use yield instead of
return
os.remove(path)

def test_something(my_files):
...
path = my_files
...
```

build in fixtures: tmp_path

- tmp_path is a build-in fixture for a path to a temporary folder

```
import pytest

@pytest.fixture
def my_files(tmp_path):
path = tmp_path / "file.txt"
shutil.copy(source,path)
yield path
# tmp_path fixture will remove itself

def test_something(my_files):
...
path = my_files
...
```

Put commonly used fixture in conftest.py

- pytest look up fixtures in conftest.py and the test module
- Put the fixtures in in conftest.py and make they will be accessible in the same test folder
- fixture name need to be unique

```
home/
so2_emission/
tests/
|-- pre_processing/
|-- |-- conftest.py
|-- |-- test_func.py  <-- can call fixture in
conftest
|-- |-- test_normalize.py  <-- can call fixture in
conftest
```

Testing Docstring Example

pytest can check all examples in docstring

```
pytest --doctest-modules
```

- uses python build in doctest feature underneath
  - doctest — Test interactive Python examples — Python 3.10.5 documentation
  - less boiler plate

```
# content of mymodule.py
def sum(list: list[float]) -> float:
    """Sum the value in a list
    Returns
    ....
    Examples
    -------------
    >>> a = [ 1.0, 2.0, 3.0]
    >>> sum(a):
    6.0
    """
```

Using alias in doctest

- doctest execute the example in its own environment and doesn't know the alias or module used in the example
- Setup the doctest namespace as a fixture in conftest.py
- set autouse to True so the fixture will be use whenever a pytest is called

```
# in conftest.py
import numpy
@pytest.fixture(autouse=True)
def add_np(doctest_namespace):
    doctest_namespace["np"] =
numpy
```

```
# in module.py
import numpy as np
def arange():
    """
    >>> a = np.arange(10)
    >>> len(a)
    10
    """
    pass
```

Use pytest.approx in doctest

- floating point error exists when comparing float in doctest
- When doctest_optionsflags = NUMBER is on
- The numbers are compared using pytest.approx() with relative tolerance equal to the precision.

```
>>> math.pi
3.14
```

- For example, the following output would only need to match to 2 decimal places when comparing 3.14 to pytest.approx(math.pi, rel=10**-2)

```
home/
so2_emission/
tests/
pytest.ini   <-- config file for
pytest
|-- pre_processing/
|-- |-- conftest.py
|-- |-- test_func.py
|-- |-- test_normalize.py
```

```
# in pytest.ini
[pytest]
doctest_optionflags = NUMBER
```

Summary

- Packages are building blocks of Python software development
- Automated testing equals money because saves time and improves software quality
- Documentation reduces communication time
- Automated package discovery improves our reach

Optional Exercise

- Available on github repo
- yungchidanielcho/corning-merck-python-workshop-2022: A python workshop for Data Science Symposium 2022 (github.com)
- github.com/yungchidanielcho/corning-merck-python-workshop-2022

Thank you