# Writing Modules and Testing

Lukas Hager

2024-05-01

# Learning Objectives

- Understand how to write classes, modules, and packages in Python
- Able to create a comprehensive package of modules
- Be familiar with the use of unit testing and how to do so in Python

# Class

# What is a Class?

- Broadly: an object constructor (sometimes people think of it as a blueprint)
- The object has attributes and methods
- Attributes:
  - Data associated with the class
- Methods:
  - Functions associated with the class

# Why?

- Bundles functionality and attributes
- DRY
- Object-Oriented Programming
  - Low-level languages are based on functions and logic
  - OOP is based on creating objects that have data and functions

# Example

- Cars
  - Two cars are different, but will share attributes (both have wheels, steering wheels, brakes, etc.)
  - The object would be a car, and we can fill in the specific attributes later
- Models
  - Two regression models are different, but share attributes (both produce coefficients, standard errors, fitted values)
  - The object would be the *model*, and we can fill in the specific attributes later (the data)

# Creating a Class

Convention: classes get SnakeCase names

```
class LinearRegression:
    def __init__(self):
        return None
```

We have a (kind of useless) class!

# Creating a Class

Convention: classes get SnakeCase names

```
class LinearRegression:
    def __init__(self):
        return None
```

We have a (kind of useless) class!

```
my_obj = LinearRegression()
```

# `__init__()`

This code runs whenever the object is created, and always takes at least one argument (`self`). Here, we'll give our regression a name so that we can keep track of different specifications. This argument will become an attribute of the class within `__init__()`:

```
class LinearRegression:
    def __init__(self, name = 'ols'):
        self.name = name
```

# `__init__()`

This code runs whenever the object is created, and always takes at least one argument (`self`). Here, we'll give our regression a name so that we can keep track of different specifications. This argument will become an attribute of the class within `__init__()`:

```
class LinearRegression:
    def __init__(self, name = 'ols'):
        self.name = name
```

We can access this attribute once the object is created:

```
my_obj = LinearRegression('panel_reg')
my_obj.name
```

```
'panel_reg'
```

# Adding a Method

Emulating `sklearn`, let's add a method for fitting a linear regression, and add the `X` and `y` data as attributes. We do this like so:

```
class LinearRegression:
    def __init__(self, name):
        self.name = name
    def fit(self, X: np.array, y: np.array):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        self.beta = np.linalg.inv(self.X.T @ self.X) @ self.X
```

# Trying out `fit`

```
rng = np.random.default_rng(seed=481)
X = rng.standard_normal(size=(1000,5))
epsilon = rng.standard_normal(size=(1000,1))
y = X @ np.array([3., 5., -2, 6., 1.5]) + epsilon

my_obj = LinearRegression('panel_reg')
my_obj.fit(X, y)
my_obj.beta
```

```
array([[-0.02338684,  0.01298824, -0.00793553, ..., -0.02349427,
        -0.01361545, -0.00545721],
       [-0.08753511, -0.01996627, -0.05883339, ..., -0.08773467,
        -0.06938418, -0.05422977],
       [ 0.07353455, -0.01142486,  0.03744571, ...,  0.07378548,
         0.05071202,  0.03165723],
       [-0.4028155 ,  0.23644813, -0.13127072, ..., -0.40470355,
        -0.23109093, -0.08771625],
       [ 0.30115639, -0.14433381,  0.11192219, ...,  0.30247214,
         0.18148493,  0.08156994]])
```

# Trying out `fit`

```
rng = np.random.default_rng(seed=481)
X = rng.standard_normal(size=(1000,5))
epsilon = rng.standard_normal(size=(1000,1))
y = X @ np.array([3., 5., -2, 6., 1.5]) + epsilon

my_obj = LinearRegression('panel_reg')
my_obj.fit(X, y)
my_obj.beta
```

```
array([[-0.02338684,  0.01298824, -0.00793553, ..., -0.02349427,
        -0.01361545, -0.00545721],
       [-0.08753511, -0.01996627, -0.05883339, ..., -0.08773467,
        -0.06938418, -0.05422977],
       [ 0.07353455, -0.01142486,  0.03744571, ...,  0.07378548,
         0.05071202,  0.03165723],
       [-0.4028155 ,  0.23644813, -0.13127072, ..., -0.40470355,
        -0.23109093, -0.08771625],
       [ 0.30115639, -0.14433381,  0.11192219, ...,  0.30247214,
         0.18148493,  0.08156994]])
```

# Dimensions

```
my_obj.X.shape, my_obj.y.shape, my_obj.beta.shape
```

```
((1000, 5), (1000, 1000), (5, 1000))
```

# Exercise: Input Sanitization

Write a method for our class to ensure that `X` and `y` make sense dimensionally before computing `beta`

# Solution: Input Sanitization

```
class LinearRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
    def fit(self, X: np.array, y: np.array):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
        self.beta = np.linalg.inv(self.X.T @ self.X) @ self.X

my_obj = LinearRegression('panel_reg')
my_obj.fit(X, y)
```

```
ValueError: Dimensions are not correct
```

# Why `_check_dims`?

- We'd consider this method "private"
  - Users should not need to call `_check_dims`
- As such, we put an underscore at the beginning to let them know that they can ignore it

# Trying Method with Correct Dimensions

```
rng = np.random.default_rng(seed=481)
X = rng.standard_normal(size=(1000,5))
epsilon = rng.standard_normal(size=(1000,1))
y = X @ np.array([3., 5., -2, 6., 1.5]).reshape(-1,1) + epsil

my_obj = LinearRegression('panel_reg')
my_obj.fit(X, y)
my_obj.beta
```

```
array([[ 3.00064876],
       [ 4.95711242],
       [-1.98260416],
       [ 6.01959136],
       [ 1.50678939]])
```

# Adding SEs to Fit

```
class LinearRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
    def fit(self, X: np.array, y: np.array) -> np.array:
        """Fit a linear regression"""
        self.X = X
        self.y = y
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
        self.beta = np.linalg.inv(self.X.T @ self.X) @ self.X
        self.resid = (self.y-self.X @ self.beta).flatten()
        self.cov = np.linalg.inv(self.X.T@self.X) * np.inner(s
```

# Adding a summary Method

```
class LinearRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
    def fit(self, X: np.array, y: np.array):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
        self.beta = np.linalg.inv(self.X.T @ self.X) @ self.X
        self.resid = (self.y-self.X @ self.beta).flatten()
        self.cov = np.linalg.inv(self.X.T@self.X) * np.inner(s
    def summary(self) -> pd.DataFrame:
        """Produce a regression table"""
        ses = np.sqrt(np.diag(self.cov))
        data = {
            'coef': self.beta.flatten(),
```

# Trying our summary Method

```
my_obj = LinearRegression('panel_reg')
my_obj.fit(X, y)
my_obj.summary()
```

| | coef | se | t-stat |
|---|---|---|---|
| 0 | 3.000649 | 0.971147 | 3.089799 |
| 1 | 4.957112 | 0.962300 | 5.151319 |
| 2 | -1.982604 | 0.964765 | -2.055013 |
| 3 | 6.019591 | 0.973967 | 6.180491 |
| 4 | 1.506789 | 0.968623 | 1.555599 |

# Exercise: Plot

Please add a plotting method to our class that shows actual data against fitted data. Include in the title the $R^2$ value, calculated as

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$$

where $\overline{y}$ is the sample mean of the $y_i$ values.

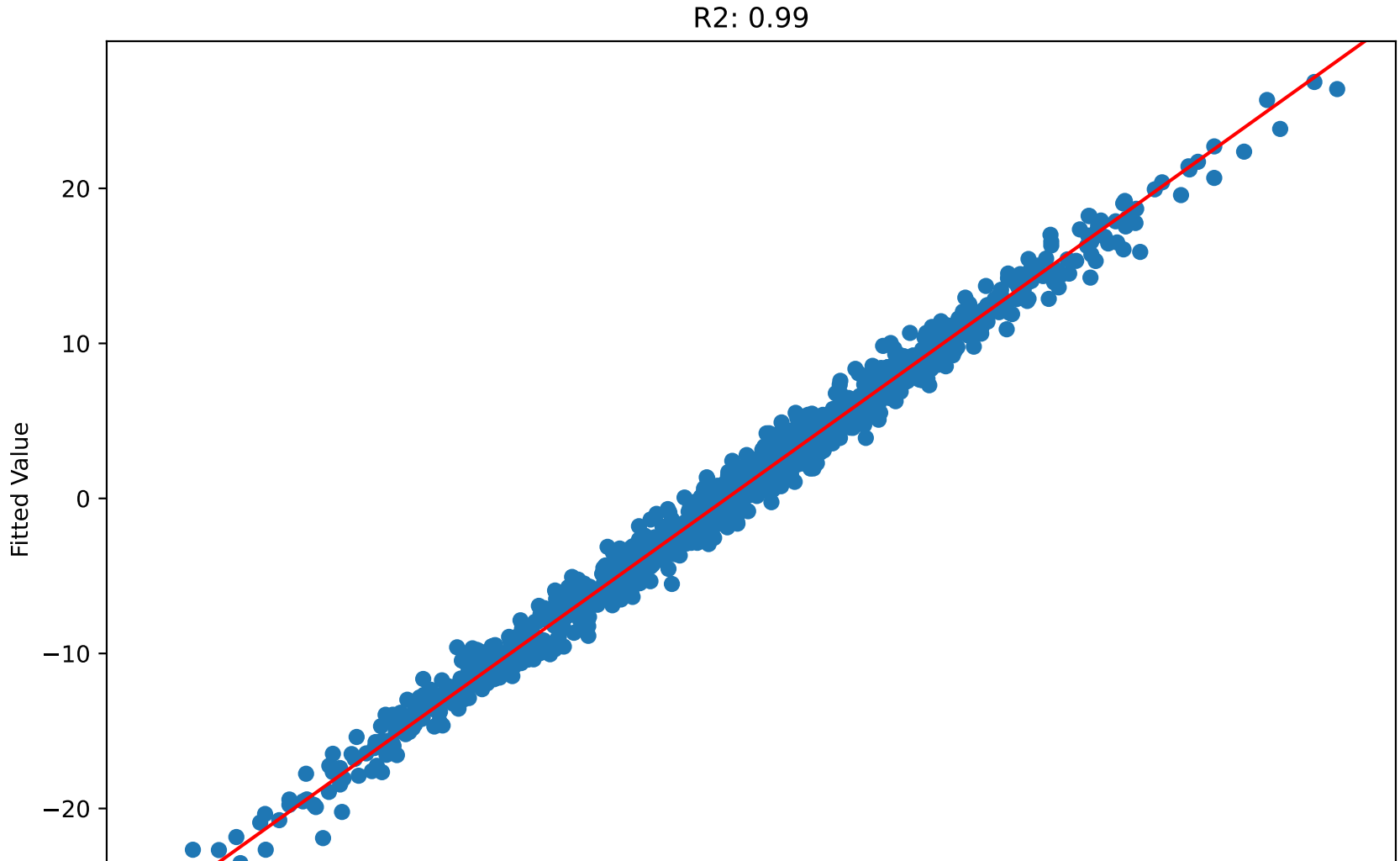
# Solutions: Plot (Class)

```
from matplotlib import pyplot as plt

class LinearRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
    def fit(self, X: np.array, y: np.array):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
        self.beta = np.linalg.inv(self.X.T @ self.X) @ self.X
        self.resid = (self.y-self.X@self.beta).flatten()
        self.cov = np.linalg.inv(self.X.T@self.X) * np.inner(s
    def summary(self) -> pd.DataFrame:
        """Produce a regression table"""
        ses = np.sqrt(np.diag(self.cov))
```

# Solutions: Plot (Use)

```
my_reg = LinearRegression('ols')
my_reg.fit(X,y)
my_reg.plot()
```



# Packages

# linear_regression.py

Let's call this a completed module for our package pyedpyper which will implement statistical methods:

**Listing 1** pyedpyper/linear_regression.py

```
import pandas as pd
import numpy as np

class LinearRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
    def fit(self, X: np.array, y: np.array):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
```

# Ridge Regression

- Recall LASSO, where we regularized a regression for prediction
- Ridge does the same thing, but with a different regularization penalty
  - LASSO penalizes the sum of the absolute values of the coefficients
  - Ridge penalizes the sum of the squared coefficients

$$\min_{\hat{\beta}} \sum_{i=1}^{n} (y - \mathbb{X}\hat{\beta})^2 + \alpha\hat{\beta}^2$$

# Ridge Solution

Ridge is nice because it has a closed-form solution:

$$\widehat{\beta} = \left(\mathbb{X}^\top \mathbb{X} + \alpha \mathbb{I}\right)^{-1} \mathbb{X}^\top y$$

# Adding `ridge_regression.py`

We can adapt our existing code pretty easily.

```
class RidgeRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
            and isinstance(self.alpha, (float,int))
    def fit(self, X: np.array, y: np.array, alpha: float):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        self.alpha = alpha
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
        denom = np.linalg.inv(self.X.T @ self.X + self.alpha
        num = self.X.T @ self.y
        self.beta = denom @ num
    def summary(self) -> pd.DataFrame:
        """Produce a regression table"""
```

# Trying Our Class

```
ridge_obj = RidgeRegression('ridge')
ridge_obj.fit(X,y,50)
ridge_obj.summary()
```

| | coef |
|---|---|
| 0 | 2.869855 |
| 1 | 4.736071 |
| 2 | -1.898266 |
| 3 | 5.737582 |
| 4 | 1.450930 |

# ridge_regression.py

**Listing 2** ridge_regression.py

```
import pandas as pd
import numpy as np

class RidgeRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
            and isinstance(self.alpha, (float,int))
    def fit(self, X: np.array, y: np.array, alpha: float):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        self.alpha = alpha
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
        denom = np.linalg.inv(self.X.T @ self.X + self.alpha
```

# Creating a Package

- It would be nice if we could run something like

```
from pyedpyper import linear_regression, ridge_regression
```

- To do this, we need to arrange the files properly in a folder and add an `__init__.py` file

# `__init__.py`

- This file marks that a directory is a package, so python should look for modules within the folder
- It can be empty – really just tells python that it should look within a folder for code

# File Structure

- `pyedpyper/`
  - `__init__.py`
  - `models/`
    - `__init__.py`
    - `linear_regression.py`
    - `ridge_regression.py`

Import:

```
from pyedpyper.models import linear_regression, ridge_regressi
```

# Notes

- The package needs to be added to the PATH global variable – can do this like so:

```
import sys
sys.path.append('/my/filepath/to/package')
```

- We import the file names without the extension (`linear_regression.py` becomes `linear_regression`)
- We can then reference the classes we've created within the file:

```
my_obj = linear_regression.LinearRegression('ols')
```

# Problem

- To create `RidgeRegression` we just copy-pasted a ton of code from `LinearRegression`
- We want to avoid repeating ourselves when possible
- How can we do this?

# utils.py

A common module that contains functions that are used by different pieces of code, generally somewhere it's easily accessible by other modules.

# Example

We could put this function in `utils.py`

```
def summary(**kwargs):
    """Produce a regression table"""
    return pd.DataFrame(kwargs)
```

# Aside: kwargs

We might not know which keyword (kw) arguments (args) will be passed by a user, or we'd like to leave all options available – we can do this by having **kwargs be a function argument (essentially an unpacked dictionary) that we use in the function.

```
def my_fun(**kwargs):
    print(type(kwargs))
    for name in kwargs.keys():
        print(f'name: {name}, value: {kwargs[name]}')

my_fun(this = 'is', a = 'test')
```

```
<class 'dict'>
name: this, value: is
name: a, value: test
```

# New File Structure

- `pyedpyper/`
  - `__init__.py`
  - `utils.py`
  - `models/`
    - `__init__.py`
    - `linear_regression.py`
    - `ridge_regression.py`

# utils.py

**Listing 3** `utils.py`

```
import pandas as pd

def summary(**kwargs):
    """Produce a regression table"""
    return pd.DataFrame(kwargs)
```

# New linear_regression.py

**Listing 4** linear_regression.py

```
import pandas as pd
import numpy as np

from .. import utils as ut

class LinearRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
    def fit(self, X: np.array, y: np.array):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        if not self._check_dims():
            raise ValueError('Dimensions are not correct')
        self.beta = np.linalg.inv(self.X.T @ self.X) @ self.X
```

# New ridge_regression.py

**Listing 5** ridge_regression.py

```
import pandas as pd
import numpy as np

from .. import utils as ut

class RidgeRegression:
    def __init__(self, name):
        self.name = name
    def _check_dims(self):
        """Check input dimensions"""
        return self.X.shape[0] == self.y.shape[0] and self.y.s
            and isinstance(self.alpha, (float,int))
    def fit(self, X: np.array, y: np.array, alpha: float):
        """Fit a linear regression"""
        self.X = X
        self.y = y
        self.alpha = alpha
        if not self._check_dims():
```

# Installation

- This is a little bit tedious – we were forced to manually add our package to the path.
- It would be better if we could just install the package where the rest of our packages are installed.
- We can do this pretty easily using `setup.py`

# `setup.py`

- Tells `pip` how to install the package from GitHub
- Lists the package dependencies
- Allows others to use your package

# Example of setup.py

**Listing 6** setup.py

```
from setuptools import setup, find_packages

setup(
    name='pyedpyper',
    version=0.01,

    url='https://github.com/lukas-hager/pyedpyper',
    author='Lukas Hager',
    author_email='lghhager@uw.edu',

    install_requires = [
        'numpy',
        'pandas',
        'matplotlib'
    ],

    packages=find_packages(),
)
```

# GitHub Site

- The code for this package can be found here
- To install via pip, we need to copy the HTTPS link (click on the green "Code" button for the link)
- Install via

```
pip install git+<your_github_https_url_here.git>
```

# GitHub Site

- The code for this package can be found here
- To install via pip, we need to copy the HTTPS link (click on the green "Code" button for the link)
- Install via

```
pip install git+<your_github_https_url_here.git>
```

In this case:

```
pip install git+https://github.com/lukas-hager/pyedpyper.git
```

# GitHub Site

- The code for this package can be found here
- To install via pip, we need to copy the HTTPS link (click on the green "Code" button for the link)
- Install via

```
pip install git+<your_github_https_url_here.git>
```

In this case:

```
pip install git+https://github.com/lukas-hager/pyedpyper.git
```

NB: You may need to use `pip3` instead of `pip` depending on your system configuration

# Testing

# Why Test?

- One reason:
  - If you have a lot of code, it can be difficult to make edits and ensure things still run properly
  - You want to feel confident when you push the edit to a subpart of the code that the rest of the code still works as anticipated
- Another reason:
  - You have a live model in production that you want to make sure hasn't "drifted"
  - That is, the data-generating process that the model was trained off of has not changed

# Common Workflow

- Develop code and tests simultaneously
  - For example, every time you write a method, write a test for that method
- Whenever you add to the codebase or change things, ensure that the code still passes the tests
- If you don't pass, you're alerted to some issue that you can fix before putting your work into the real world

# Alternative to Unit Testing

# Common Testing Framework – `pytest`

- We should add testing to our package – `pytest` makes this easy to do.
- We can define a set of tests and run them to make sure our package still works

# Folder Structure Reminder

Our directory looks like this

- `pyedpyper/`
  - `__init__.py`
  - `models/`
    - `__init__.py`
    - `linear_regression.py`
    - `ridge_regression.py`

# Adding Tests

Our new directory will look like this

- `pyedpyper/`
  - `__init__.py`
  - `models/`
    - `__init__.py`
    - `linear_regression.py`
    - `ridge_regression.py`
  - `tests/`

# Exercise: Test Ideas

Recall our linear and ridge regression modules. Think of what kind of tests we could run to make sure that we get correct results. In this case, think about "ground truth" results – what are some datasets we could pass our code where we "know" the results?

# Solutions: Test Ideas

- If we have wholly deterministic data, we should be able to recover the regression coefficient exactly
- If we pass data in that has the wrong dimensions, we should raise a specific error (the one that we wrote)
- Our coefficients should match the coefficients that are generated by a package like `statsmodels`
- If we specify a regularization parameter of 0, we should get the same coefficients from OLS and Ridge

# Test Coverage

- Formally defined as the percentage of code in our package that's unit tested
- We want to get as close to 100% as we can – that indicates that there aren't chunks of code that have no checks
- Ideally, we'd write a check for all of our methods in our classes

# Writing Tests

- `pytest` will look for tests within files that are prefixed by "test"
- The tests are functions that contain statements like `assert` that will error out if the asserted condition is not met

# Example of a Test

Suppose we had a function like this

```
def addition(a: float, b: float) -> float:
    """Add a and b"""
    return a + b
```

# Example of a Test

Suppose we had a function like this

```
def addition(a: float, b: float) -> float:
    """Add a and b"""
    return a + b
```

We could have a test for it:

```
def test_addition():
    assert addition(3, 5) == 8
```

# Example of a Test

Suppose we had a function like this

```
def addition(a: float, b: float) -> float:
    """Add a and b"""
    return a + b
```

We could have a test for it:

```
def test_addition():
    assert addition(3, 5) == 8
```

Under the hood, pytest will run the test:

```
test_addition()
```

# Example of a Test

Suppose we had a function like this

```
def addition(a: float, b: float) -> float:
    """Add a and b"""
    return a + b
```

We could have a test for it:

```
def test_addition():
    assert addition(3, 5) == 8
```

Under the hood, pytest will run the test:

```
test_addition()
```

No errors

# How to Run pytest

```
pytest <filename if you want to test a specific file>
```

# Testing Deterministic Model

```
rng = np.random.default_rng()

def test_fit_no_error():
    X = rng.randn(1000, 5)
    beta = np.arange(1.,6.).reshape(-1,1)
    y = X @ beta
    lr = LinearRegression('ols')
    lr.fit(X,y)
    assert np.allclose(lr.beta,beta,rtol=0,atol=1e-6)
```

- `atol`: absolute tolerance
- `rtol`: relative tolerance
- condition: `abs(a - b) <= (atol + rtol * abs(b))`

# Testing Comparison

```
def test_comparison():
    X = rng.randn(1000, 5)
    beta = np.arange(1.,6.).reshape(-1,1)
    y = X @ beta
    lr = LinearRegression('ols')
    lr.fit(X,y)
    rr = RidgeRegression('ridge')
    rr.fit(X,y,0)
    assert np.allclose(lr.beta,rr.beta,rtol=0,atol=1e-6)
```

# Testing Error

```
import pytest

def test_error_lr():
    X = rng.randn(1000, 5)
    y1 = rng.randn(1001, 1)
    y2 = rng.randn(1000, 2)
    lr = LinearRegression()
    with pytest.raises(ValueError):
        lr.fit(X,y1)
    with pytest.raises(ValueError):
        lr.fit(X,y2)

def test_error_rr():
    X = rng.randn(1000, 5)
    y1 = rng.randn(1001, 1)
    y2 = rng.randn(1000, 2)
    rr = RidgeRegression()
    with pytest.raises(ValueError):
        rr.fit(X,y1)
    with pytest.raises(ValueError):
```

# Creating Test Files

- We can store the tests for each module in their own file within our `tests` folder
- We then have the ability to run `pytest` on our module

# Regression Tests

**Listing 7** `test_linear_regression.py`

```
import pytest
import numpy as np

from pyedpyper.models.linear_regression import LinearRegressi
from pyedpyper.models.ridge_regression import RidgeRegression

rng = np.random.default_rng()

def test_fit_no_error():
    X = rng.random((1000, 5))
    beta = np.arange(1.,6.).reshape(-1,1)
    y = X @ beta
    lr = LinearRegression('ols')
    lr.fit(X,y)
    assert np.allclose(lr.beta,beta,rtol=.001,atol=0)

def test_comparison():
    X = rng.random((1000, 5))
```

# Ridge Tests

```
import pytest
import numpy as np

from pyedpyper.models.ridge_regression import RidgeRegression

rng = np.random.default_rng()

def test_error_rr():
    X = rng.random((1000, 5))
    y1 = rng.random((1001, 1))
    y2 = rng.random((1000, 2))
    rr = RidgeRegression('ridge')
    with pytest.raises(ValueError):
        rr.fit(X,y1,1)
    with pytest.raises(ValueError):
        rr.fit(X,y2,1)
```

# Testing

Within the package folder we can run `pytest` to get the test results:

```
============================================================
platform darwin -- Python 3.8.5, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/hlukas/git/pyedpyper
plugins: csv-3.0.0
collected 4 items

tests/test_linear_regression.py ...
tests/test_ridge_regression.py .
```

# Exercise: Testing a `DataFrame`

Look at the documentation for pd.testing.assert_frame_equal and use it to write a unit test for our `utils.summary` method.

# Solutions: Testing a `DataFrame`

```
def test_summary():
    df1 = ut.summary(a=np.arange(6), b=np.arange(6,12))
    df2 = pd.DataFrame({'a': np.arange(6), 'b': np.arange(6,1
    pd.testing.assert_frame_equal(df1, df2)
```

# utils Test File

**Listing 8** test_utils.py

```
import pandas as pd
import numpy as np

from pyedpyper import utils as ut

def test_summary():
    df1 = ut.summary(a=np.arange(6), b=np.arange(6,12))
    df2 = pd.DataFrame({'a': np.arange(6), 'b': np.arange(6,1
    pd.testing.assert_frame_equal(df1, df2)
```

# Testing

Again, we can run this with `pytest`:

```
============================================================
platform darwin -- Python 3.8.5, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/hlukas/git/pyedpyper
plugins: csv-3.0.0
collected 5 items

tests/test_linear_regression.py ...
tests/test_ridge_regression.py .
tests/test_utils.py .
```

# Code Coverage

- A useful metric is how much of our code is unit tested
- For example, passing all tests is much more relevant when those tests are testing every method in our codebase compared to when they're just testing a single function

# Installation of `pytest-cov`

- `pytest` has the capability to provide this to us directly if we install a second package

```
pip install pytest-cov
```

# Usage of pytest-cov

```
pytest --cov

============================================================
platform darwin -- Python 3.8.5, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/hlukas/git/pyedpyper
plugins: cov-5.0.0, csv-3.0.0
collected 5 items

tests/test_linear_regression.py ...
tests/test_ridge_regression.py .
tests/test_utils.py .

---------- coverage: platform darwin, python 3.8.5-final-0 -----
Name                                      Stmts   Miss  Cover
-------------------------------------------------------------
pyedpyper/models/__init__.py                  0      0   100%
pyedpyper/models/linear_regression.py        19      2    89%
pyedpyper/models/ridge_regression.py         19      1    95%
pyedpyper/utils.py                            3      0   100%
tests/__init__.py                             0      0   100%
tests/test_linear_regression.py              30      0   100%
```

# Pre-Commit Hooks

# Why?

- We want high-quality commits
  - For example, we want the code to confirm to a stylistic standard
  - We want certain types of files to be committed together
- Humans are forgetful and it's hard to go through a lot of code to make sure it's all consistent
- We'll add something to our package that will check our commits when we submit them

# Installation

```
pip install pre-commit
```

# YAML

We'll add a file called `.pre-commit-config.yaml` to our package that tells `pre-commit` what we want it to look for (this is the baseline file that can be found here):

**Listing 9** `.pre-commit-config.yaml`

```
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v2.3.0
    hooks:
    -   id: check-yaml
    -   id: end-of-file-fixer
    -   id: trailing-whitespace
-   repo: https://github.com/psf/black
    rev: 22.10.0
    hooks:
    -   id: black
```

For example, this will (stylistically) remove trailing whitespace and add a blank line at the end of files.

# YAML: Aside

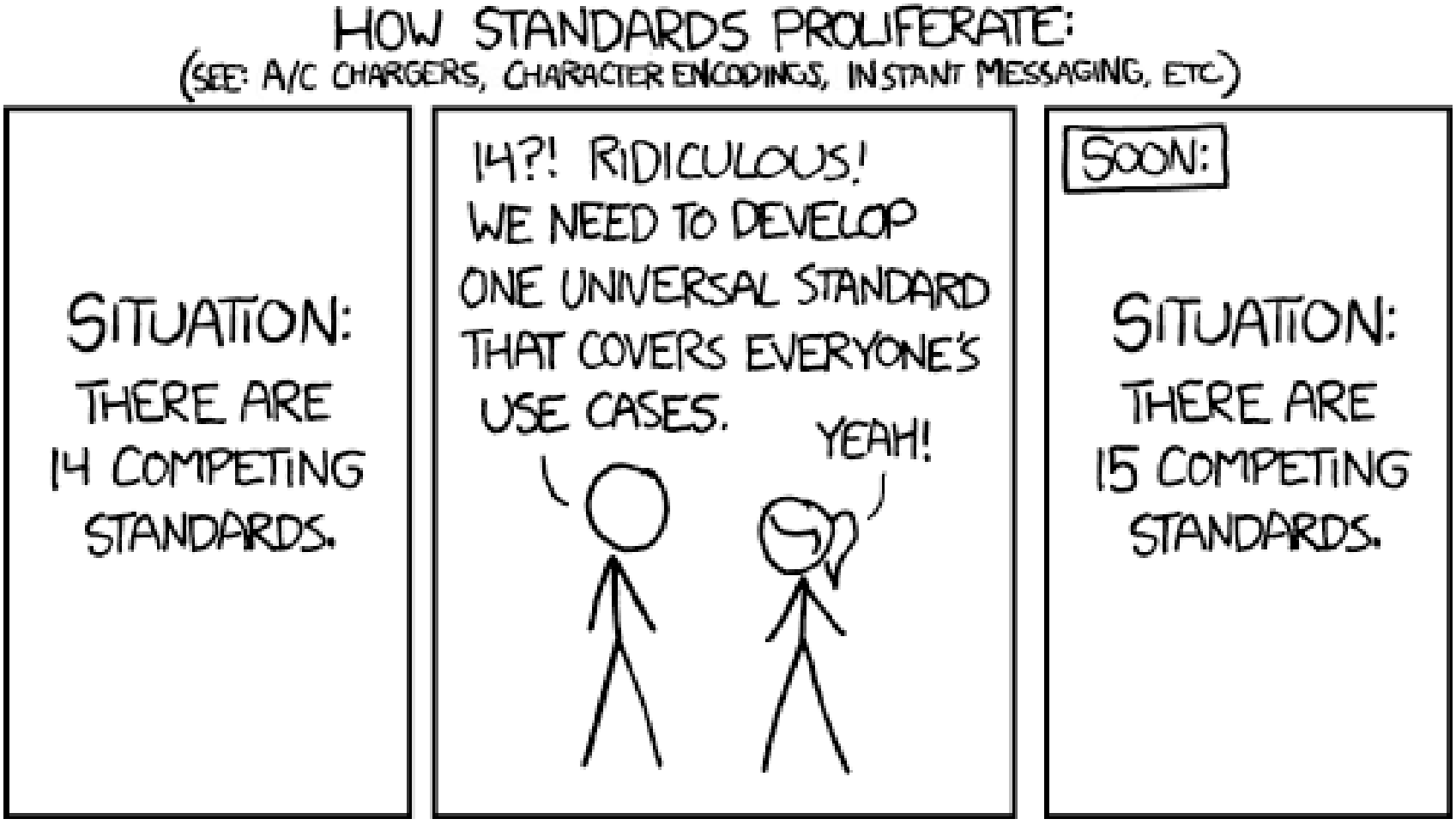



Figure 2: YAML stands for "Yet Another Markup Language"

# Installing

We can run

```
pre-commit install
```

to set up our new hooks

# Commit Output

```
Check Yaml...........................................
Fix End of Files.....................................
- hook id: end-of-file-fixer
- exit code: 1
- files were modified by this hook

Fixing .pre-commit-config.yaml
Fixing tests/test_utils.py
Fixing tests/test_linear_regression.py
Fixing tests/test_ridge_regression.py

Trim Trailing Whitespace.....................................
black................................................
- hook id: black
- files were modified by this hook

reformatted tests/test_utils.py
reformatted tests/test_ridge_regression.py
reformatted tests/test_linear_regression.py
```

# Linting

Remember that we can lint our code to make it conform stylistically – this is something that we can add to our pre-commit hook

**Listing 10** `.pre-commit-config.yaml`

```
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v2.3.0
    hooks:
    -   id: check-yaml
    -   id: end-of-file-fixer
    -   id: trailing-whitespace
-   repo: https://github.com/psf/black
    rev: 22.10.0
    hooks:
    -   id: black
-   repo: https://github.com/pylint-dev/pylint
    rev: v3.1.0
    hooks:
    -   id: pylint
```

# Running with Linting

We get output like this:

```
************* Module tests.test_ridge_regression
tests/test_ridge_regression.py:1:0: C0114: Missing module docstr
tests/test_ridge_regression.py:9:0: C0116: Missing function or m
tests/test_ridge_regression.py:10:4: C0103: Variable name "X" do
```

# Virtual Environments

# Why?

- We want to limit the number of issues we run into due to differences in package verisons
- We want to make sure that the correct packages are installed while developing

# What?

- A version of python with specific versions of packages installed
- Anyone can create this version of python
- Ensures that things are standardized
- I generally run them through Anaconda

# environment.yml

A file that lists out the dependencies for our virtual environment:

**Listing 11** `environment.yml`

```
name: pyedpyper
dependencies:
  - numpy
  - pandas
  - pre-commit
  - pylint
```

- Can be created via `conda env create -f environment.yml`
- Can be activated via `conda activate pyedpyper`

# pyedpyper

The repository with all of these files can be found here