UNI FREIBURG

Python Course

WT 24/25

Lecture: Marcus Rockel & Prof. Dr. Eva Lütkebohmert-Holtz, Tutorial: Felix Häschel
https://www.finance.uni-freiburg.de/

Tutorial 13

Exercise 1 (Population Data)**.**
The UN provides two sheets ("Estimates" and "Medium variant") in a file found on Ilias in the data folder called `UN_2024_Population_Data.xlsx`. Since the file is a bit larger, and reading from excel can be slow, you have the option to instead use the provided parquet files `UN_2024_Population_Data_Estimates.parquet` and `UN_2024_Population_Data_Medium_variant.parquet` instead.
Complete the following:

- Read each sheet, skip the irrelevant top rows, rename the first valid row as headers, rename ``Region, subregion, country or area *`` to simply "`Region`," and keep only ``Region,`` ``Year,`` and the columns for ages `0--99` plus "100+".
- Sum across those age columns to produce ``Total Population``.
- Pivot the table so that each row is a **Year** and each column is a **Region** (the values being the total population).
- Concatenate the two sheets (e.g. "Estimates" + "Medium variant") into a single dataset, multiply by **1,000** if needed (the data is in thousands).
- Produce a **plot** (line or bar) for selected regions (e.g. `World, China, India`) across years.
- Aggregate the population data by region and income levels across decades:
  Use the classifications provided in `income_levels.csv` and `oecd_countries.csv` to group countries into regions. Compute the average population for each region across decades, starting from a specified start year. Generate a bar plot showing the mean population for each region per decade to visualize trends.

Solution. The script with the solution has been pushed to our GitHub repository.

Exercise 2 (GNCM Model)**.**
In our GitHub repository is a folder called `macroeconomics`, which contains a file called `3_GNCM_Model_1.py`. It is a Python script from Prof. Dr. Hillebrand that implements the GNCM model, which is a macroeconomic model discussed in his course.
Help Prof. Dr. Hillebrand speed up his code by approximately 25 % by refactoring no more than six (consecutive) lines of code (you will not need to understand the GNCM model or what the code exactly does).

Solution. The issue in the code lies in the lambda function on line 173, where `eta_x` and `F` are repeatedly recalculated within the `Phi` function. These are computationally expensive operations that do not change throughout the iterations. A more efficient approach would be to calculate these values once outside the lambda function and reuse them. Additionally, the use of the `jit` decorator from the `numba` library can further speed up the code by applying just-in-time compilation.

```
@jit
def compute_X(K, N, v_old, S_1_old, S_2_old, h, e):
    # Precompute constants outside the Phi function to avoid redundant calculations
    eta_x_val = eta_x(K, N*h, e)
    F_val = F(K, N*h, e)

    def Phi(X):
        return eta_x_val * F_val / X - \
        alpha * (1 - eta_x_val) * F_val * F_val * (v_old - c_x) / \
        (1 - D(S_1_old + S_2_old * (1 - phi) + phi * X)) - tau_bar * F_val

    # Find a number X_max at which Phi(X_max)<0:
    X_max = .1
    phi_max = 1
    while phi_max >0:
        X_max = X_max*2
        phi_max = Phi(X_max)
    # Minimum values at which Phi(X_max)>0:
    X_min = X_max/2
    phi_val = 1
    counter = 0
    # Determine X by bisecting the interval [X_min, X_max]:
    while abs(phi_val) > err_tol and counter < 10**5:
        X = (X_max+X_min)/2
        phi_val = Phi(X)
        if phi_val >0:
            X_min = X
        if phi_val <0:
            X_max = X
        counter +=1
    return X
```

Exercise 3 (Black)**.**
Format the following code snippet in PyCharm using Black and take note of its changes:

```
import unittest
def add(
    a, b
):
    return a + b
class TestAdd(unittest.TestCase):
    '''
    Test the add function
    '''
    def test_add_positive(self):
        self.assertEqual(add(2, 3), 5)
    def test_add_negative(self):
        self.assertEqual(add(-1, -1), -2)
if __name__ == "__main__":
    unittest.main()
```

Solution. After formatting the code with Black, several changes were applied to make it adhere strictly to PEP 8 standards. In the original code, the function definition `def add(a, b):` was unnecessarily spread across multiple lines, which Black reformatted into a single, more concise line. Similarly, the spacing and alignment of the class docstring and methods were adjusted to ensure consistent indentation and blank lines as per PEP 8 guidelines. Black enforces uniformity in code style, such as maintaining two blank lines before the class definition and avoiding redundant line breaks. Additionally, Black changed the single to double quotes. These changes improve readability, streamline the code, and ensure it follows Python's official style guide.

Exercise 4 (Catch the bug)**.**
Find the failing test case in the `portfolio_analytics` package and catch the error without explicit use of break points. Fix the test.

Solution. The problem lies in the value for the volatility (`0.06`) in the assertions of the `test_portfolio_performance()` function. The hardcoded value is incorrect, leading to a failing test. Below is the corrected test case where the correct values are assigned to variables before calculating the differences.

```
def test_portfolio_performance():
    mean_ret = pd.Series({"A": 0.1, "B": 0.05})
    cov = pd.DataFrame(
        data={"A": [0.04, 0.00], "B": [0.00, 0.02]}, index=["A", "B"]
    )
    weights = np.array([0.6, 0.4])
    port_ret, port_vol = portfolio_performance(weights, mean_ret, cov)

    # Assign the correct expected values to variables
    expected_return = 0.6 * 0.1 + 0.4 * 0.05  # 0.6*0.1 + 0.4*0.05 = 0.08
    expected_volatility = (0.6**2 * 0.04 + 0.4**2 * 0.02)**0.5  # sqrt(0.6^2 * 0.04 +
    ↪  0.4^2 * 0.02) ~ 0.13266499161

    # Calculate differences
    return_difference = abs(port_ret - expected_return)
    volatility_difference = abs(port_vol - expected_volatility)

    assert return_difference < 1e-9
    assert volatility_difference < 1e-9
```

This approach ensures that the hardcoded values are replaced with dynamically calculated ones, making the test case accurate and robust.

Exercise 5 (Test coverage)**.**
Create an HTML report for the test coverage of the `portfolio_analytics` package in our GitHub repository.

Solution. An HTML test coverage report was generated using `pytest-cov`. Below are the commands executed to generate the report (assuming your directory in the terminal is `portfolio-analytics`):

```
pytest --cov=portfolio_analytics --cov-report=html
```

The HTML report is located in the `htmlcov` folder. Open `htmlcov/index.html` in a browser to view the detailed coverage report.

Exercise 6 (Parallel testing)**.**
Run the tests in the `portfolio_analytics` package in parallel with up to four workers.

Solution. Parallel testing was conducted using `pytest-xdist`. The command used to run tests with 4 workers is as follows:

```
pytest -n 4
```

You can adjust the number of workers used for parallel testing, or use `pytest -n auto` to perform testing with a number of workers processes equal to the number of available CPUs. This ensures faster test execution by running tests concurrently instead of sequentially. For a low number of tests sequential execution can be faster, since the workers used for parallel testing have to be initialized first.