Numpy

Kittikun Jitpairod

Introduction to NumPy in Economics

NumPy (Numerical Python) is a fundamental package for scientific computing in Python. Its application in economics is widespread due to its efficient handling of large datasets and its comprehensive mathematical functions. This section explores the utilization of NumPy for basic economic calculations.

Installing and Importing NumPy

Before proceeding with calculations, it is necessary to install and import NumPy:

```
# Installation (execute in command line)
# pip install numpy

# Importing in Python script
import numpy as np
```

Creating Arrays for Economic Data

NumPy's primary object is the homogeneous multidimensional array. In economic contexts, these arrays often represent time series data, price vectors, or matrices of economic indicators.

```
# Creating a 1D array of prices
prices = np.array([100, 102, 104, 103, 106])

# Creating a 2D array of supply and demand quantities
supply_demand = np.array([[100, 120, 140], [120, 110, 100]])

print(prices)
print(supply_demand)
```

```
[100 102 104 103 106]
[[100 120 140]
 [120 110 100]]
```

Basic Statistical Calculations

Economic analysis often requires descriptive statistics to summarize data. NumPy provides functions for essential statistical calculations. Let's compare NumPy implementations with standard Python implementations:

Mean (Average)

The mean represents the central tendency of a dataset.

Equation: $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$

Standard Python implementation:

```
prices = [100, 102, 104, 103, 106]
mean_price_py = sum(prices) / len(prices)
print(mean_price_py)
```

```
103.0
```

NumPy implementation:

```
mean_price_np = np.mean(prices)
print(mean_price_np)
```

```
103.0
```

Median

The median is the middle value when a dataset is ordered from least to greatest. It's useful for understanding the central tendency when data is skewed.

Standard Python implementation:

```
sorted_prices = sorted(prices)
n = len(sorted_prices)
if n % 2 == 0: # even number
    median_price_py = (sorted_prices[n//2 - 1] + sorted_prices[n//2]) / 2
else: # odd number
    median_price_py = sorted_prices[n//2]
print(median_price_py)
```

```
103
```

NumPy implementation:

```
median_price_np = np.median(prices)
print(median_price_np)
```

```
103.0
```

Standard Deviation

Standard deviation measures the amount of variation or dispersion of a set of values. In economics, it's used to quantify the amount of variation or dispersion of a set of data values.

Equation: $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$

Standard Python implementation:

```
mean = sum(prices) / len(prices)

squared_diff = [
    (x - mean) ** 2 for x in prices]

variance = sum(squared_diff) / len(prices)

price_std_py = variance ** 0.5

print(price_std_py)
```

```
2.0
```

NumPy implementation (1):

```
mean = np.sum(prices) / len(prices)

squared_diff = (prices - mean) ** 2

variance = np.sum(squared_diff) / len(prices)

price_std_manual_np = np.sqrt(variance)

print(price_std_manual_np)
```

```
2.0
```

NumPy implementation (2):

```
price_std_np = np.std(prices)
print(price_std_np)
```

```
2.0
```

Comparison and Discussion

As demonstrated, NumPy provides more concise and readable implementations for these statistical calculations. The standard Python implementations require more lines of code and explicit loop constructs.

Key advantages of using NumPy for these calculations include:

1. Conciseness: NumPy functions like `np.mean()`, `np.median()`, and `np.std()` encapsulate complex operations in single function calls.
2. Performance: NumPy operations are implemented in C, making them significantly faster than equivalent Python loops, especially for large datasets.
3. Precision: NumPy uses optimized algorithms that can provide better numerical stability and precision, particularly for operations like standard deviation on large datasets.
4. Vectorization: NumPy allows for vectorized operations on entire arrays, which is more efficient and easier to read and maintain than explicit loops.
5. Consistency: NumPy provides a consistent interface for working with multi-dimensional data, which is particularly useful when dealing with more complex economic datasets.

Time Value of Money Calculations

The time value of money is a core principle in finance and economics. NumPy's exponential and financial functions facilitate these calculations:

Future Value

Future Value (FV) calculates the value of a current asset at a future date based on an assumed growth rate.

Equation: $FV = PV \cdot e^{rt}$

Where:

- FV = Future Value
- PV = Present Value
- r = interest rate (as a decimal)

- t = number of time periods

```
def FV(PV:float, rate:float, time:int) -> float:
    return PV * np.exp(rate * time)

print(FV(1000.0, 0.05, 5))
```

```
1284.0254166877414
```

Matrix Operations for Input-Output Analysis

Input-Output analysis in economics often involves matrix operations, which NumPy handles efficiently. This analysis examines the interdependencies between different sectors of the economy.

Leontief Inverse

The Leontief inverse is crucial in input-output analysis. It's used to compute the total output required to meet a given final demand.

Equation: $X = (I - A)^{-1} \cdot D$

Where:

- X = Total output vector
- I = Identity matrix
- A = Matrix of technical coefficients
- D = Final demand vector

```
A = np.array([[0.2, 0.3], [0.4, 0.1]])
D = np.array([[100], [200]])
I = np.eye(A.shape[0])

print('Matrix A')
print(A)

print('\nMatrix D')
print(D)

print('\nMatrix I')
print(I)

print('\n')
print(np.linalg.inv(I - A) @ D)
print(np.matmul(np.linalg.inv(I - A), D))
```

```
Matrix A
[[0.2 0.3]
 [0.4 0.1]]

Matrix D
[[100]
 [200]]

Matrix I
[[1. 0.]
 [0. 1.]]


[[250.        ]
 [333.33333333]]
[[250.        ]
 [333.33333333]]
```

- `np.eye()`: create an identity matrix
- `A.shape`: get the dimension of A
- `np.linalg`: linear algebra module
- `@` operator: matrix multiplication, equivalent to `np.matmul(A, B)`

Calculating Economic Indicators

NumPy can be used to calculate various economic indicators:

GDP Growth Rate

The GDP growth rate measures how fast the economy is growing. It's typically calculated on an annual basis.

Equation: GDP Growth Rate $= \frac{\text{GDP}_t - \text{GDP}_{t-1}}{\text{GDP}_{t-1}} \times 100\%$

```python
gdp_values = np.array([100, 102, 105, 108, 110])

gdp_growth_rates = (gdp_values[1:] - gdp_values[:-1]) / gdp_values[:-1] * 100

print(gdp_growth_rates)
```

```
[2.         2.94117647 2.85714286 1.85185185]
```

Inflation Rate

The inflation rate measures the rate at which the general level of prices for goods and services is rising, consequently eroding purchasing power.

Equation: Inflation Rate $= \frac{\text{CPI}_t - \text{CPI}_{t-1}}{\text{CPI}_{t-1}} \times 100\%$

```
cpi_values = np.array([200, 204, 209, 213, 218])

inflation_rates = (cpi_values[1:] - cpi_values[:-1]) / cpi_values[:-1] * 100

print(inflation_rates)
```

```
[2.         2.45098039 1.9138756  2.34741784]
```

Optimization Problems

Many economic problems involve optimization. NumPy, in conjunction with SciPy, can solve these problems efficiently.

Cost Minimization (more info.)

This example demonstrates a simple cost minimization problem subject to a constraint.

Objective function: $\min f(x) = x_1^2 + x_2^2$

Subject to: $x_1 + x_2 = 1$

```
from scipy.optimize import minimize

def cost(x):
    return x[0]**2 + x[1]**2

def constraint(x):
    return x[0] + x[1] - 1

result = minimize(cost, [0.5, 0.5], method='SLSQP',
            constraints={'type': 'eq', 'fun': constraint})

print(result)
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
```

```
 fun: 0.5
   x: [ 5.000e-01  5.000e-01]
 nit: 1
 jac: [ 1.000e+00  1.000e+00]
nfev: 3
njev: 1
```