FN6815_Lect_03_Numpy

February 27, 2024

FN6815 Numerical Methods for Financial Instrument Pricing

Lecture 3: Python numeric programming

- Dr. Yang Ye
- Email: yy@runchee.com
- 2023/2024 Mini Term 5

1. Numpy

1.1 Why Numpy?

The base data type in Python is list. It is not efficient for numerical programming and it lacks of mathematical tools.

The NumPy library has equipped Python as a powerful tool for numerical programming.

It is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

- **n-D Arrays**: NumPy's core feature is the n-dimensional array (`np.array`). This data type supports vector and matrix operations. For example, adding two arrays is as simple as `np.array([1, 2]) + np.array([1, 2])`, which results in `array([2, 4])`.
- **Mathematical Functions**: NumPy provides a host of mathematical functions for operations like exponentiation, logarithms, random number generation, averages, and statistical distributions. These functions are designed to work with n-D arrays.
- **Performance**: NumPy leverages C for numerical computations, which significantly improves performance.

1.2 Numpy/Scipy/Matplotlib setup

1. Use `conda install numpy scipy matplotlib`, if haven't done so.
2. import in Python. Write following lines on top of each Jupyter notebook for `numpy and friends`.

[1]:
```python
import matplotlib.pyplot as plt  # plotting
import numpy as np
import scipy as sp  # additional numerical tools
```

if you saw something prompt like below, use `conda install missing package`.

Usually, we import the numpy library with the alias `np`. Every time we use a function/object from numpy, we will use `np.function_name`.

```
[2]: import numpy as np

     np.array([1, 2]) + np.array([1, 2])
```

```
[2]: array([2, 4])
```

1.3 Comparison NumPy v.s. Python

Let's use the following example to compare the performance of NumPy and Python.

The example is a program of random walker. A walker starts at 0 and at each step, it moves 1 step to the left or right with equal probability. The program simulates the walker's position after `n` steps.

1. Python Native

```
[3]: import random


     class RandomWalker:
         def __init__(self):
             self.position = 0

         def walk(self, n):
             self.position = 0
             for i in range(n):
                 yield self.position
                 self.position += 2 * random.randint(0, 1) - 1


     walker = RandomWalker()
     %timeit -n10 -r10 walk = [position for position in walker.walk(10000)]
```

```
4.3 ms ± 55 µs per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

2. Full vectorization via numpy

```
[4]: import numpy as np


     def random_walk_fastest(n):
         # No 's' in numpy choice (Python offers choice & choices)
         steps = np.random.choice([-1, +1], n)
         return np.cumsum(steps)


     %timeit -n10 -r10 walk = random_walk_fastest(10000)
```

```
65.6 µs ± 6.55 µs per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

The return of function `random_walk_fastest(10000)` is an `nd-array` of 10000 elements. The nd-array is an object with many mathematical methods. For example, we can calculate the mean and standard deviation of the nd-array using `np.mean` and `np.std`. This avoid the use of for loop and is much faster than the Python native version.

Using numpy means that we use `nd-array` as the basic data type for numbers (though nd-array can store other data types, too). When we write code to deal with numbers, we shall make nd-array as the input and output as much as possible.

This is the key to achieve high performance in numerical programming.

Note

We shall see numpy's gain in performance as space exchange for efficiency. The nd-array is a data type that is not as flexible as Python's list. For example, we cannot append an element to an nd-array. We have to create a new nd-array with the appended element. This is a trade-off between performance and flexibility.

The **flat** memory model of nd-array makes the performance gain possible. The flat memory model means that the elements of an nd-array are stored in a continuous memory block.

1.4 Numpy's ndarray Class and Array Creation Functions

Numpy's core component is the N-dimensional array, or ndarray. This data structure allows efficient operations on large datasets. You can refer to the Numpy documentation for more details.

Bookmark the help page of Numpy: https://numpy.org/doc/stable/reference/index.html

Here are several ways to create ndarray:

1. `np.array([])`: This function creates an array from a list or tuple.

```
# Create vector
a = np.array([0, 0.5, 1.0, 1.5, 2.0, 3.5])

# Create 2-d array
a = np.array([[4, 3, 5], [1, 2, 1]])

# Create 2-d array from one-d
b = a.reshape(6, 1) # (2, 3) (1, 6)
```

2. `np.arange(start, stop, step)` and `np.linspace(start, stop, num)`: These functions generate sequences of numbers in an array. arange uses a step value, while linspace uses the total number of items.

```
a = np.arange(0, 10, 2) # array([0, 2, 4, 6, 8])
b = np.linspace(0, 10, 3) # array([0., 5., 10.]) of 3 elements
```

3. `np.ones(shape)`, `np.zeros(shape)`: These functions create arrays filled with ones or zeros, respectively.

```
a = np.ones((3, 3)) # 3x3 array of ones
b = np.zeros((2, 2)) # 2x2 array of zeros
```

4. `.reshape(newshape)`: Be aware of how reshape() works. Numpy usually takes the row-major order. So `.reshape()` breaks the array to the new rows.

```
a = np.arange(6)
b = a.reshape(2, 3) # array([[0, 1, 2], [3, 4, 5]])
```

```
[5]: (
         np.array(range(1, 5)).reshape(4, 1),
         np.array(range(1, 5)).reshape(1, 4),
         np.array(range(1, 5)).reshape(2, 2),
     )
```

```
[5]: (array([[1],
             [2],
             [3],
             [4]]),
      array([[1, 2, 3, 4]]),
      array([[1, 2],
             [3, 4]]))
```

1.5 Numpy Mini-reference

1. `np_array.size == 0`
2. `.ndim`, `.shape`

```
array2x3 = np.array(
    [
        [1, 2, 3],
        [9, 8, 7],
    ]
)

array2x3.ndim # 2
array2x3.shape # (2,3)
```

3. sort: `b = np.sort(a, axis=1)`

```
np.sort([2,1])
```

4. `np.sum`, `max()`, `sum()`, `cumsum()`, `std()`

```
np.sum([1, 2, 3]) # 6
np.sum([[1, 2, 3], [1, 2, 3]]) # 12
np.sum([[1, 2, 3], [1, 2, 3], [1, 2, 3],], axis=0)

# these functions are also ndarray's method. call it with the variable.
a = np.array([1, 2, 3])
a.sum()
```

5. Mathematical operations, `+, -, *, **, /, >, <` are overloaded to apply them to np.ndarray.

```
y = x ** 2
```

6. Mathematical functions, `np.exp/np.log`
7. Slicing:

   ```
   # This is derivatives' numeric form.

   dy_dx = (y[1:]-y[:-1])/(x[1:]-x[:-1])
   dy_dx_c = (y[2:]-y[:-2])/(x[2:]-x[:-2])
   ```

8. random:

   ```
   import numpy.random as npr
   points = npr.random((100000, 3))
   npr.uniform((1, 3))
   N = 10
   p = npr.choice(N, N)
   ```

As you shall see, many common mathematical operations are overloaded to apply to np.ndarray. Other functions are re-created in the numpy, instead of math, for `sin`, `log`, etc. This makes the code to take nd-array as input and output naturally.

Additional example of combining *(melting), slicing and recursion together

Let's illustrate how to convert a tuple of dimensions obtained from `np.shape` into a single number representing the total number of elements, similar to `np.size`.

Here's a Python function that uses tuple unpacking (also known as "melting"), slicing, and recursion to achieve this:

In this example, multiply_dimensions recursively multiplies all the dimensions together. It starts with the first dimension and multiplies it by the result of the function called on the rest of the tuple, effectively "melting" the tuple down to a single number.

The base case for the recursion is an empty tuple, which signifies that all dimensions have been processed. " " "

[6]:
```python
import numpy as np


def multiply_dimensions(nd):
    def _internal(*a):
        # print("_internal:", a)
        if len(a) == 0:
            return 1
        return a[0] * _internal(*a[1:])

    return _internal(*nd.shape)


mat = np.array([[1, 2, 3], [2, 3, 4]])
```

```python
print(multiply_dimensions(mat))
```

```
6
```

1.6 Caution: Passing data between functions

In Python, objects such as lists, dictionaries, Numpy arrays (np.ndarray), and Pandas DataFrames are passed by *reference*.

Objects can be thought of as containers for data. As a beginner, it's advisable to adopt a style where a function takes an object as input and returns a new object, rather than modifying the original object. This should be your default approach. Such function that has no side effect (i.e. do not modify the existing object) is called *pure function*.

Here are some key points to remember:

1. Avoid modifying variables passed into a function. If you need to alter the data, create a copy first to prevent side effects outside the function. This may introduce some performance overhead, but it's crucial to prioritize correct results before optimization.

[7]:
```python
# impure function
def n_attacks_impure(p):
    p[p > 1] = 0  # This updates p.
    return p.sum()


input_data = np.array([7.0, 7.0, 0.0, 1.0, 7.0, 2.0, 1.0, 3.0])
print(
    "This n_attacks_impure() changed the input:",
    n_attacks_impure(input_data),
    input_data,
)


# pure function
def n_attacks(p):
    pv = p.copy()  # make a copy here
    pv[pv > 1] = 0  # change the copy not the original.
    return pv.sum()


# For some cases, we can avoid copying the array
def n_attacks_better(p):
    return p[p <= 1].sum()


input_data = np.array([7.0, 7.0, 0.0, 1.0, 7.0, 2.0, 1.0, 3.0])
print(
    "This n_attacks() pure version does not change the input:",
    (n_attacks(input_data), n_attacks_better(input_data)),
    input_data,
)
```

```
This n_attacks_impure() changed the input: 2.0 [0. 0. 0. 1. 0. 0. 1. 0.]
This n_attacks() pure version does not change the input: (2.0, 2.0) [7. 7. 0. 1.
7. 2. 1. 3.]
```

2. In certain scenarios, you might want to modify the input object directly within a function. This approach is contrary to the previously discussed style of returning a new object. To achieve this, you can use `.update` or similar methods provided by many Python objects.

[8]:
```
def update_it_bad(ll):
    ll = (
        [-1] + ll
    )  # This will not update the original object because this is an assignment␣
 ↪to a local variable `ll`. It's of the same name but different object.
    return sum(ll)


input_data = [1, 2, 3]
print(
    "This update_it_bad() *did not* update the input:",
    update_it_bad(input_data),
    input_data,
)


def update_it(ll):
    ll.insert(0, -1)  # this is an update method.
    return sum(ll)


input_data = [1, 2, 3]
print("This update_it() version changes the input:", update_it(input_data),␣
 ↪input_data)
```

```
This update_it_bad() *did not* update the input: 5 [1, 2, 3]
This update_it() version changes the input: 5 [-1, 1, 2, 3]
```

2. Numpy Examples

2.1 Black-Scholes Formula

Numpy: Efficient Array Operations Numpy is designed to work with array data, enabling efficient operations on all elements of an array in parallel. This feature is often referred to as "vectorization".

Here's an example:

[9]:
```
np.array(range(10)) + 100
```

[9]:
```
array([100, 101, 102, 103, 104, 105, 106, 107, 108, 109])
```

[10]:
```
from scipy.stats import norm


def bs_call(S, X, T, r, sigma):
    d1 = (np.log(S / X) + (r + sigma * sigma / 2.0) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * norm.cdf(d1) - X * np.exp(-r * T) * norm.cdf(d2)


c = bs_call(40, 42, 0.5, 0.015, 0.2)
print(c)

c1 = bs_call(40, 43, 0.5, 0.015, 0.1)
print(c1)

# Numpy can apply to vector
print(
    bs_call(
        np.array([40, 40]),
        np.array([42, 43]),
        np.array([0.5, 0.5]),
        np.array([0.015, 0.015]),
        np.array([0.2, 0.1]),
    )
)

# map is to apply the function to all elements.
print(list(map(np.asarray, zip([40, 42, 0.5, 0.015, 0.2], [40, 43, 0.5, 0.015,␣
 ↪0.1]))))

# Use the * to expand list, equivalent to [np.asarray(x) for x in [...]]
print(
    bs_call(*map(np.asarray, zip([40, 42, 0.5, 0.015, 0.2], [40, 43, 0.5, 0.
 ↪015, 0.1])))
)
```

```
1.556578806289311
0.28443591964924586
[1.55657881 0.28443592]
[array([40, 40]), array([42, 43]), array([0.5, 0.5]), array([0.015, 0.015]),
array([0.2, 0.1])]
[1.55657881 0.28443592]
```

2.2 Broadcasting

Broadcasting: Expanding Dimensions of Input Data Broadcasting is a powerful mechanism in Numpy that allows mathematical operations to be performed between arrays of different shapes.

Consider the following example that calculates the inter-milepost distances:

In this example, `mileposts[:, np.newaxis]` adds an extra dimension to the mileposts array, transforming it from a 1D array into a 2D array. This allows the subtraction operation to be broadcasted across all combinations of milepost locations, resulting in a matrix of inter-milepost distances.

[11]:
```
mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544])

print(mileposts)
print(mileposts[:, np.newaxis])  # to create column-vector

inter_distance = np.abs(mileposts - mileposts[:, np.newaxis])
print(inter_distance)
```

```
[   0  198  303  736  871 1175 1475 1544]
[[   0]
 [ 198]
 [ 303]
 [ 736]
 [ 871]
 [1175]
 [1475]
 [1544]]
[[   0  198  303  736  871 1175 1475 1544]
 [ 198    0  105  538  673  977 1277 1346]
 [ 303  105    0  433  568  872 1172 1241]
 [ 736  538  433    0  135  439  739  808]
 [ 871  673  568  135    0  304  604  673]
 [1175  977  872  439  304    0  300  369]
 [1475 1277 1172  739  604  300    0   69]
 [1544 1346 1241  808  673  369   69    0]]
```

[12]:
```
np.arange(1, 10) * np.arange(1, 10)[:, np.newaxis]
```

[12]:
```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9],
       [ 2,  4,  6,  8, 10, 12, 14, 16, 18],
       [ 3,  6,  9, 12, 15, 18, 21, 24, 27],
       [ 4,  8, 12, 16, 20, 24, 28, 32, 36],
       [ 5, 10, 15, 20, 25, 30, 35, 40, 45],
       [ 6, 12, 18, 24, 30, 36, 42, 48, 54],
       [ 7, 14, 21, 28, 35, 42, 49, 56, 63],
       [ 8, 16, 24, 32, 40, 48, 56, 64, 72],
       [ 9, 18, 27, 36, 45, 54, 63, 72, 81]])
```

[13]:
```python
x, y = np.arange(5), np.arange(5)[:, np.newaxis]
distance = np.sqrt(x**2 + y**2)
print(distance)
```

```
[[0.         1.         2.         3.         4.        ]
 [1.         1.41421356 2.23606798 3.16227766 4.12310563]
 [2.         2.23606798 2.82842712 3.60555128 4.47213595]
 [3.         3.16227766 3.60555128 4.24264069 5.        ]
 [4.         4.12310563 4.47213595 5.         5.65685425]]
```

3. Assignment

1. Pi in wallis algorithm

The Wallis formula is an infinite product that converges to Pi/2. We can use it to compute the value of Pi. The calculation stops when the difference between successive approximations is less than 1e-9.

Implement it with Python and Numpy. Compare the performance of the two implementations.

Reference

2. Quicksort

Quicksort is a popular sorting algorithm. Here's a Python implementation based on the pseudo-code provided:

```
function quicksort(array)
    var list less, greater
    if length(array) < 2
        return array
    select and remove a pivot value pivot from array
    for each x in array
        if x < pivot + 1 then append x to less
        else append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

3. Exercise: crude integration

We'll write a function $f(a, b, c)$ that returns $a^b - c$, and use it to approximate a 3D integral over the parameter ranges [0, 1] x [0, 1] x [0, 1].

We can approximate the integral by forming a 24x12x6 grid over the parameter ranges and averaging the function values at the grid points.

This is to approximate the 3-d integral $\int_0^1 \int_0^1 \int_0^1 (a^b - c) \mathrm{d}a \mathrm{d}b \mathrm{d}c$ cover this volume with the mean.

The exact answer is: $ln2 - \frac{1}{2} \approx 0.1931\ldots$ — what is your relative error and time spent?

Implement it the base Python and Numpy.

Note There are at least two ways to solve this kind of problem: the conventional for loop, or numpy broadcasting.

It's good to be familiar with both approaches because some interviewer may ask "could you do it from scratch, without numpy?"

Hint: for broadcasting method 1a. You can make a 1-D grid with `np.linspace(start,end,grids)`, apply `[:, np.newaxis]` to obtain the 2nd axis, apply `[:, np.newaxis][:, np.newaxis]` to obtain the 3rd axis.

1b. Alternative to 1a, `i,j,k = np.ogrid[0:1:24j, 0:1:12j, 0:1:6j]` is a shortcut to step 1a.

2. Run $f(a, b, c)$ with a, b, c on different axis would use broadcasting to obtain results for all combinations of a, b, c

Appendix: timestamp

[14]:
```
from datetime import datetime

print(f"Generated on {datetime.now()}")
```

```
Generated on 2024-02-27 21:11:35.203636
```