23. More on 2D Arrays

A Class with 2D Array Attributes
More Practice with 2D Array OPs
More Practice with numpy

The Setting

A company has $m$ factories and each of which makes $n$ products. We'll refer to such a company as an $m$-by-$n$ company.

Customers submit purchase orders in which they indicate how many of each product they wish to purchase. A length-$n$ list of numbers that expresses this called a PO list.

Cost and Inventory

The cost of making a product varies from factory to factory.

Inventory varies from factory to factory.

Three Problems

A customer submits a purchase order that is to be filled by a single factory.

Q1. How much would it cost each factory to fill the PO?
Q2. Which factories have enough inventory to fill the PO?
Q3. Among the factories that can fill the PO, which one can do it most cheaply?

Ingredients

To set ourselves up for the solution to these problems we need to understand:

- The idea of a Cost Array (2D)
- The idea of an Inventory Array (2D)
- The idea of a Purchase Order Array (1D)

We will use numpy arrays throughout.

Cost Array

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>36</th>
<th>22</th>
<th>15</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>35</td>
<td>20</td>
<td>12</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>37</td>
<td>21</td>
<td>16</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

The value of $C[k,j]$ is what it costs factory $k$ to make product $j$. 
**Cost Array**

The value of \( c_{k,j} \) is what it costs factory \( k \) to make product \( j \).

- \( c_{1,3} = 12 \) to make product 3 for factory 1.

**Inventory Array**

The value of \( i_{k,j} \) is the inventory in factory \( k \) of product \( j \).

- Factory 1 can sell up to 83 units of product 2.

**Purchase Order**

The value of \( po_j \) is the number of product \( j \)’s that the customer wants.

- Customer wishes to purchase 29 units of product 3.

**We Will Develop a Class called Company**

We will package data and methods in a way that makes it easy to answer Q1, Q2, and Q3 and to perform related computations.
First, Some Handy Numpy Features

Computing Row and Column Dimension

Suppose:

\[
I = \begin{bmatrix}
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{bmatrix}
\]

A 2-by-3 array.

\[
I = \text{array}([[10,36,22],[12,35,20]])
\]

Computing Row and Column Dimension Using shape

Suppose:

\[
I = \begin{bmatrix}
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{bmatrix}
\]

\[(m,n) = I.\text{shape}\]

\[
(m,n) = \text{I.shape} \quad m: 2 \quad n: 3
\]

Useful in functions and methods with 2D array arguments

Finding the Location of the Smallest Value Using argmin

>>> from numpy import *
>>> x = array([20,40,10,70,60])
>>> iMin = x.argmin()
>>> xMin = x[iMin]
>>> print iMin, xMin
2 10

There is also an argmax method

Comparing Arrays

>>> x = array([20,10,30])
>>> y = array([2,1,3])
>>> z = array([10,40,15])

>>> x>y
array([ True,  True,  True], dtype=bool)

>>> all(x>y)
True

>>> x>z
array([ True, False,  True], dtype=bool)

>>> any(x>z)
True

>>> 1/x
array([0.05, 0.25, 0.2], dtype=float64)

>>> x+1
array([21, 41, 31], dtype=int64)

>>> x+1
array([ Inf, 40, 15], dtype=float64)

inf

A special float that behaves like infinity

>>> x = inf
>>> 1/x
0
>>> x+1
Inf
>>> inf > 9999999999999
True
Now Let's Develop the Class Company

Start with the attributes and the constructor.

```
class Company:
    """
    Attributes:
    C : m-by-n cost array [float]
    I : m-by-n inventory array [float]
    TV : total value [float]
    """

    Total Value: How much is the total inventory worth ?
```

The Class Company: Constructor

```
def __init__(self, Inventory, Cost):
    self.I = Inventory
    self.C = Cost
    (m,n) = Inventory.shape
    TV = 0
    for k in range(m):
        for j in range(n):
            TV += Inventory[k,j]*Cost[k,j]
    self.TV = TV
```

The incoming arguments are the Inventory and Cost Arrays

Row and Column Dimensions

```
def __init__(self, Inventory, Cost):
    self.I = Inventory
    self.C = Cost
    (m,n) = Inventory.shape
    TV = 0
    for k in range(m):
        for j in range(n):
            TV += Inventory[k,j]*Cost[k,j]
    self.TV = TV
```

To compute the row and column dimension of a numpy 2D array, use the shape attribute.

Computing Total Value

```
TV = 0
for k in range(m):
    for j in range(n):
        TV += I[k,j]*C[k,j]
```

```
I --> | 10 36 22 |
     | 12 35 20 |
C --> | 30 40 50 |
      | 60 70 80 |
```

Inventory Array  Cost Array

Computing Total Value

```
TV = 0
for k in range(m):
    for j in range(n):
        TV += I[k,j]*C[k,j]
```

```
I --> | 10 36 22 |
     | 12 35 20 |
C --> | 30 40 50 |
      | 60 70 80 |
```

Inventory Array  Cost Array
Computing Total Value

\[ TV = 0 \]
\[
\text{for } k \text{ in range}(m): \\
\quad \text{for } j \text{ in range}(n): \\
\quad \quad TV += I[k,j] * C[k,j]
\]

\[
\begin{array}{c|c}
I & \text{Inventory Array} \\
\hline
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{array}
\]

\[
\begin{array}{c|c}
C & \text{Cost Array} \\
\hline
30 & 40 & 50 \\
60 & 70 & 80 \\
\end{array}
\]

Computing Total Value

\[ TV = 0 \]
\[
\text{for } k \text{ in range}(m): \\
\quad \text{for } j \text{ in range}(n): \\
\quad \quad TV += I[k,j] * C[k,j]
\]

\[
\begin{array}{c|c}
I & \text{Inventory Array} \\
\hline
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{array}
\]

\[
\begin{array}{c|c}
C & \text{Cost Array} \\
\hline
30 & 40 & 50 \\
60 & 70 & 80 \\
\end{array}
\]

Computing Total Value

\[ TV = 0 \]
\[
\text{for } k \text{ in range}(m): \\
\quad \text{for } j \text{ in range}(n): \\
\quad \quad TV += I[k,j] * C[k,j]
\]

\[
\begin{array}{c|c}
I & \text{Inventory Array} \\
\hline
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{array}
\]

\[
\begin{array}{c|c}
C & \text{Cost Array} \\
\hline
30 & 40 & 50 \\
60 & 70 & 80 \\
\end{array}
\]

Computing Total Value

\[ TV = 0 \]
\[
\text{for } k \text{ in range}(m): \\
\quad \text{for } j \text{ in range}(n): \\
\quad \quad TV += I[k,j] * C[k,j]
\]

\[
\begin{array}{c|c}
I & \text{Inventory Array} \\
\hline
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{array}
\]

\[
\begin{array}{c|c}
C & \text{Cost Array} \\
\hline
30 & 40 & 50 \\
60 & 70 & 80 \\
\end{array}
\]

Now Let's Develop Methods

to Answer These 3 Questions

Q1. How much would it cost each factory to fill a purchase order?

Q2. Which factories have enough inventory to fill a purchase order?

Q3. Among the factories that can fill the purchase order, which one can do it most cheaply?
Q1. How Much Does it Cost Each Factory to Process a Purchase order?

For factory 0:

\[
s = 0; 
\text{for } j \in \text{range}(5): 
    s = s + C[0,j] \times PO[j]
\]

1*10 + 0*36 + 12*22 + 29*15 + 5*62
For factory 0:
```
s = 0;
for j in range(5):
s += C[0,j] * PO[j]
```

For factory 1:
```
s = 0;
for j in range(5):
s += C[1,j] * PO[j]
```

To Answer Q1 We Have
```
def Order(self,PO):
    """ Returns an m-by-1 array that houses how much it costs each factory to fill the PO."
    """
    PreC: self is a Company object representing m factories and n products. PO is a length-n purchase order list.
    """
    def Order(self,PO):
        C = self.C
        (m,n) = C.shape
        theCosts = zeros((m,1))
        for k in range(m):
            for j in range(n):
                theCosts[k] += C[k,j]*PO[j]
        return theCosts
```

What the Order Method Does
```
self.C -->
10  36  22  15  62
12  35  20  12  66
13  37  21  16  59

PO -->
1  0  12  29  5

Returns [1019, 930, 1040]
```

Implementation...
```
def Order(self,PO):
    C = self.C
    (m,n) = C.shape
    theCosts = zeros((m,1))
    for k in range(m):
        for j in range(n):
            theCosts[k] += C[k,j]*PO[j]
    return theCosts
```
Using Order

Assume that the following are initialized:

- `I` the Inventory array
- `C` the Cost array
- `PO` the purchase order array

```python
>>> A = Company(I, C)
>>> x = A.Order(PO)
>>> kMin = x.argmin()
>>> xMin = x[kMin]
```

`kMin` is the index of the factory that can most cheaply process the `PO` and `xMin` is the cost.

### Q2. Which Factories Have Enough Inventory to Process a Purchase Order?

#### Who Can Fill the Purchase Order (PO)?

<table>
<thead>
<tr>
<th></th>
<th>38</th>
<th>5</th>
<th>99</th>
<th>34</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>82</td>
<td>19</td>
<td>83</td>
<td>12</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>29</td>
<td>21</td>
<td>56</td>
<td>87</td>
</tr>
<tr>
<td>PO</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
</tr>
</tbody>
</table>

Factory 2 can’t because 12 < 29

#### Who Can Fill the Purchase Order (PO)?

<table>
<thead>
<tr>
<th></th>
<th>38</th>
<th>5</th>
<th>99</th>
<th>34</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>82</td>
<td>19</td>
<td>83</td>
<td>12</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>29</td>
<td>21</td>
<td>56</td>
<td>87</td>
</tr>
<tr>
<td>PO</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
</tr>
</tbody>
</table>

`all(I[0, :] >= PO)` is True

<table>
<thead>
<tr>
<th></th>
<th>38</th>
<th>5</th>
<th>99</th>
<th>34</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>82</td>
<td>19</td>
<td>83</td>
<td>12</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>29</td>
<td>21</td>
<td>56</td>
<td>87</td>
</tr>
<tr>
<td>PO</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
</tr>
</tbody>
</table>

`all(I[1, :] >= PO)` is False
Who Can Fill the Purchase Order (PO)?

I --> 38 5 99 34 42
     82 19 83 12 42
     51 29 21 56 87
PO --> 1 0 12 29 5

all(I[2,:]>=PO) is True

To Answer Q2 We Have...

def CanDo(self,PO):
    """ Return the indices of those factories with sufficient inventory. """
    PreC: PO is a purchase order array. """

Who Can Fill the PO?

def CanDo(self,PO):
    I = self.I
    (m,n) = I.shape
    Who = []
    for k in range(m):
        if all( I[k,:] >= PO ):
            Who.append(k)
    return array(Who)

Who Can Fill the PO?

def CanDo(self,PO):
    I = self.I
    (m,n) = I.shape
    Who = []
    for k in range(m):
        if all( I[k,:] >= PO ):
            Who.append(k)
    return array(Who)

Who Can Fill the PO?

def CanDo(self,PO):
    I = self.I
    (m,n) = I.shape
    Who = []
    for k in range(m):
        if all( I[k,:] >= PO ):
            Who.append(k)
    return array(Who)

Who Can Fill the PO?

def CanDo(self,PO):
    I = self.I
    (m,n) = I.shape
    Who = []
    for k in range(m):
        if all( I[k,:] >= PO ):
            Who.append(k)
    return array(Who)

Who Can Fill the PO?

def CanDo(self,PO):
    I = self.I
    (m,n) = I.shape
    Who = []
    for k in range(m):
        if all( I[k,:] >= PO ):
            Who.append(k)
    return array(Who)
Using CanDo

Assume that the following are initialized:
- I  the Inventory array
- C  the Cost array
- PO the purchase order array

```python
>>> A = Company(I, C)
>>> kVals = A.CanDo(PO)
```

Q3: Among the Factories with enough Inventory, who can fill the PO Most Cheaply??

For Q3 We Have

```python
def theCheapest(self, PO):
    """ Return the tuple (kMin, costMin) where kMin is the index of the factory
    that can fill the PO most cheaply and costMin is the associated cost. If no
    such factory exists, return None.
    PreC: PO is a purchase order list.""
    ```

Who Can Fill the Purchase Order Most Cheaply?

<table>
<thead>
<tr>
<th>I</th>
<th>38</th>
<th>5</th>
<th>99</th>
<th>34</th>
<th>42</th>
<th>Yes</th>
<th>1019</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>82</td>
<td>19</td>
<td>83</td>
<td>12</td>
<td>42</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>29</td>
<td>21</td>
<td>56</td>
<td>87</td>
<td>Yes</td>
<td>1040</td>
</tr>
</tbody>
</table>

PO ---> 1 0 12 29 5

kMin = 0,  costMin = 1019

Implementation

```python
def theCheapest(self, PO):
    theCosts = Order(PO)
    Who = CanDo(PO)
    if len(Who) == 0:
        return None
    else:
        # Find kMin and costMin
```
**Implementation Cont’d**

```python
# Find kMin and costMin
costMin = inf
for k in Who:
    if theCosts[k]<costMin:
        kMin = k
        costMin = theCosts[k]
return (kMin, costMin)
```

**Using Cheapest**

Assume that the following are initialized:
- `I` the Inventory array
- `C` the Cost array
- `PO` the purchase order array

```python
>>> A = Company(I, C)
>>> (kMin, costMin) = A.Cheapest(PO)
```

The factory with index kMin can deliver PO most cheaply and the cost is costMin.

**Updating the Inventory After Processing a PO**

<table>
<thead>
<tr>
<th>I</th>
<th>38</th>
<th>5</th>
<th>99</th>
<th>34</th>
<th>42</th>
<th>82</th>
<th>19</th>
<th>83</th>
<th>12</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>82</td>
<td>19</td>
<td>83</td>
<td>12</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>29</td>
<td>21</td>
<td>56</td>
<td>87</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>1019</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>1040</td>
<td></td>
</tr>
</tbody>
</table>

**Updating Inventory**

```python
def UpDate(self, k, PO):
    n = len(PO)
    for j in range(n):
        # Reduce the inventory of product j
        self.I[k][j] = self.I[k][j] - PO[j]
        # Decrease the total value
        self.TV = self.TV - self.C[k][j]*PO[j]
```

**Method for Updating the Inventory Array After Processing a PO**

Maintaining the class invariant, i.e., the connection between the I, C, and TV attributes.