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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>36</td>
<td>22</td>
<td>15</td>
<td>62</td>
<td></td>
</tr>
<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 $j$.

<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></td>
<td></td>
<td></td>
<td>12</td>
<td>66</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>59</td>
</tr>
</tbody>
</table>

It costs $12 for factory 1 to make product 3.
The value of $I[k,j]$ is the inventory in factory $k$ of product $j$. 
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.
The value of $PO[j]$ is the number product $j$'s that the customer wants
The value of \( PO[j] \) is the number product \( j \)'s that the customer wants.

The customer wishes to purchase 29 product 3 units.
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:

$I = \begin{array}{ccc}
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{array}$

A 2-by-3 array.

$I = \text{array}([[10,36,22],[12,35,20]])$
Computing Row and Column Dimension Using shape

Suppose:

\[
\begin{array}{ccc}
10 & 36 & 22 \\
12 & 35 & 20 \\
\end{array}
\]

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

Useful in functions and methods with 2D array arguments

\((m,n)\) is a “tuple”

shape is an attribute of the array class
Finding the Location of the Smallest Value Using argmin

```python
>>> 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

```python
>>> 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
```
inf

A special float that behaves like infinity

```python
>>> x = inf
>>> 1/x
0
>>> x+1
Inf
>>> inf > 99999999999999999999
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]
    """
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
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

\[ TV = 0 \]

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

The nested loop takes us to each array entry.

<table>
<thead>
<tr>
<th>I</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td>22</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>35</td>
<td>40</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
</tr>
</tbody>
</table>

**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]

Inventory Array

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>36</td>
<td>22</td>
</tr>
<tr>
<td>12</td>
<td>35</td>
<td>20</td>
</tr>
</tbody>
</table>

Cost Array

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>70</td>
<td>80</td>
</tr>
</tbody>
</table>
Computing Total Value

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

\[
TV = 0
\]

for \( k \) in range(m):
    for \( j \) in range(n):
        \( TV += I[k,j] \times C[k,j] \)
Computing Total Value

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

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

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

\[
\begin{align*}
\text{C} & \quad \begin{array}{cccccc}
10 & 36 & 22 & 15 & 62 \\
12 & 35 & 20 & 12 & 66 \\
13 & 37 & 21 & 16 & 59
\end{array} \\
\text{PO} & \quad \begin{array}{cccccc}
1 & 0 & 12 & 29 & 5
\end{array}
\end{align*}
\]

\[1 \times 10 + 0 \times 36 + 12 \times 22 + 29 \times 15 + 5 \times 62\]
For factory 0:

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

\[
\begin{align*}
\text{s} & = 0; \\
\text{for } j \text{ in range}(5): \\
\text{s} & = +\text{=} C[0,j] * \text{PO}[j]
\end{align*}
\]
For factory 0:

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

```python
s = 0;
for j in range(5):
    s += C[0,j] * PO[j]
```
\[ s = 0; \]
\[
\text{for } j \text{ in range}(5): \\
\quad s = += C[0,j] \times PO[j]
\]
For factory 1:

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

```python
s = 0;
for j in range(5):
    s = s + C[k,j] * PO[j]
```
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.
    """
## What the Order Method Does

The Order Method processes a list of indices and returns a new list of the specified elements. The method takes a list of indices (self.C) and returns the elements at those indices from the list (PO).

### Example

<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>

**self.C**: 1019, 930, 1040

**PO**: 1, 0, 12, 29, 5

**Returns**: [1019, 930, 1040]
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

>>> 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)?

- **I → Factory 2**: 12 < 29 → **Yes**
- **PO → Factory 1**: 12 > 29 → **No**

**Factory 2 can’t because 12 < 29**
Who Can Fill the Purchase Order (PO)?

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

We need to compare the rows of I with PO.
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>

\[
\text{all}( I[0, :] \geq PO ) \text{ is True}
\]
Who Can Fill the Purchase Order (PO)?

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

| PO | 1 | 0 | 12 | 29 | 5 |

\[ \text{all( I[1, :] \geq PO ) is False} \]
Who Can Fill the Purchase Order (PO)?

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

| 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. """
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)

Grab the inventory array and compute its row and col dimension.
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)

If every element of I[k,:] is >= the corresponding entry in PO, then factory k has sufficient inventory.
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)
```

kVals is an array that contains the indices of those factories with enough inventory
Using CanDo

Assume that the following are initialized:

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

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

If k in kVals is True, then
all(A.I[k,:]>= PO) is True
Q3: Among the Factories with enough Inventory, who can fill the PO Most Cheaply??
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></th>
<th>I</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>5</td>
</tr>
</tbody>
</table>

- kMin = 0, costMin = 1019

Yes

No

Yes

1019

1040
def theCheapest(self, PO):
    theCosts = Order(PO)
    Who = CanDo(PO)
    if len(Who) == 0:
        return None
    else:
        # Find kMin and costMin
# 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

\[
\text{>>> } A = \text{Company}(I,C)
\]
\[
\text{>>> } (kMin, \text{costMin}) = A.\text{Cheapest}(PO)
\]

The factory with index \( kMin \) can deliver \( PO \) most cheaply and the cost is \( \text{costMin} \).
Updating the Inventory After Processing a PO
## Updating Inventory

<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>

Before

Yes

No

Yes

1019

1040
# Updating Inventory

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>82</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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