25. Two-Dimensional Arrays

Topics

Motivation
The numpy Module
Subscripting functions and 2d Arrays
A 2D array has rows and columns.

This one has 3 rows and 4 columns.

We say it is a “3-by-4” array (a.k.a matrix)

Can have a 2d array of strings or objects.

But we will just deal with 2d arrays of numbers.
## Rows and Columns

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>17</td>
<td>49</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>18</td>
<td>82</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td>12</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

This is row 1.
# Rows and Columns

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>49</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>17</td>
<td></td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>18</td>
<td>82</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td>12</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

This is column 2.
Entries

This is the (1,2) entry.
Where Do They Come From?

Entry \((i,j)\) is the distance from city \(i\) to city \(j\)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amsterdam</td>
<td>0</td>
<td>650.594</td>
<td>1084.367</td>
<td>204.7</td>
<td>766.456</td>
<td>945.404</td>
<td></td>
<td>2254.519</td>
<td>476.014</td>
<td>1783.664</td>
<td>1071.746</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Berlin</td>
<td>651.304</td>
<td>0</td>
<td>1634.132</td>
<td>764.797</td>
<td>379.95</td>
<td>1506.491</td>
<td></td>
<td>2904.264</td>
<td>1036.101</td>
<td>2333.429</td>
<td>1033.506</td>
<td>502.536</td>
<td>1053.671</td>
<td>1513.741</td>
</tr>
<tr>
<td>3</td>
<td>Bordeaux</td>
<td>1064.547</td>
<td>1630.61</td>
<td>0</td>
<td>890.135</td>
<td>1785.177</td>
<td>1444.887</td>
<td></td>
<td>1117.052</td>
<td>975.717</td>
<td>703.237</td>
<td>1018.437</td>
<td>1284.774</td>
<td>562.938</td>
<td>1508.036</td>
</tr>
<tr>
<td>4</td>
<td>Brussels</td>
<td>207.37</td>
<td>767.381</td>
<td>891.028</td>
<td>0</td>
<td>908.03</td>
<td>775.414</td>
<td></td>
<td>2061.177</td>
<td>306.244</td>
<td>1690.322</td>
<td>881.246</td>
<td>784.539</td>
<td>310.51</td>
<td>1487.05</td>
</tr>
<tr>
<td>5</td>
<td>Copenhagen</td>
<td>766.376</td>
<td>381.155</td>
<td>1785.684</td>
<td>906.197</td>
<td>0</td>
<td>1645.681</td>
<td></td>
<td>2956.016</td>
<td>1177.511</td>
<td>2465.161</td>
<td>1414.722</td>
<td>1080.551</td>
<td>1205.349</td>
<td>2101.726</td>
</tr>
<tr>
<td>6</td>
<td>Dublin</td>
<td>939.70</td>
<td>1459.75</td>
<td>1439.475</td>
<td>769.049</td>
<td>1640.41</td>
<td>0</td>
<td></td>
<td>2609.627</td>
<td>453.606</td>
<td>2139.772</td>
<td>1541.326</td>
<td>1554.939</td>
<td>663.562</td>
<td>2227.14</td>
</tr>
<tr>
<td>7</td>
<td>Lisbon</td>
<td>2561.111</td>
<td>2757.07</td>
<td>1171.614</td>
<td>2066.693</td>
<td>2951.741</td>
<td>2611.451</td>
<td>0</td>
<td>2142.281</td>
<td>626.064</td>
<td>2150.156</td>
<td>2448.668</td>
<td>1749.502</td>
<td>2535.253</td>
<td>2185.753</td>
</tr>
<tr>
<td>8</td>
<td>London</td>
<td>478.973</td>
<td>1038.94</td>
<td>978.688</td>
<td>308.242</td>
<td>1179.603</td>
<td>465.078</td>
<td></td>
<td>2148.822</td>
<td>0</td>
<td>1677.965</td>
<td>1180.91</td>
<td>1094.131</td>
<td>402.746</td>
<td>1768.323</td>
</tr>
<tr>
<td>10</td>
<td>Milan</td>
<td>1074.297</td>
<td>1035.63</td>
<td>1019.438</td>
<td>905.951</td>
<td>1415.052</td>
<td>1672.432</td>
<td></td>
<td>2152.653</td>
<td>1202.042</td>
<td>1800.336</td>
<td>0</td>
<td>492.726</td>
<td>847.819</td>
<td>594.634</td>
</tr>
<tr>
<td>11</td>
<td>Munich</td>
<td>822.295</td>
<td>582.946</td>
<td>1262.355</td>
<td>763.493</td>
<td>1078.935</td>
<td>1554.472</td>
<td></td>
<td>2450.036</td>
<td>1090.302</td>
<td>1576.362</td>
<td>493.963</td>
<td>0</td>
<td>626.256</td>
<td>929.665</td>
</tr>
<tr>
<td>12</td>
<td>Paris</td>
<td>802.799</td>
<td>1046.75</td>
<td>583.226</td>
<td>308.327</td>
<td>1203.439</td>
<td>869.622</td>
<td></td>
<td>1735.377</td>
<td>400.452</td>
<td>1282.622</td>
<td>948.469</td>
<td>830.414</td>
<td>0</td>
<td>1413.908</td>
</tr>
<tr>
<td>13</td>
<td>Rome</td>
<td>1600.357</td>
<td>1514.24</td>
<td>1509.625</td>
<td>1492.011</td>
<td>1976.829</td>
<td>2257.272</td>
<td></td>
<td>2540.524</td>
<td>1708.132</td>
<td>1568.207</td>
<td>596.54</td>
<td>930.032</td>
<td>1431.299</td>
<td>0</td>
</tr>
</tbody>
</table>
Where Do they Come From?

Entry \((i,j)\) is 1 if node \(i\) is connected to node \(j\) and is 0 otherwise.

Captures the connectivity in a network.

Nodes 4 and 6 are connected.
Where Do They Come From

An m-by-n array of pixels.

Each pixel encodes 3 numbers: a red value, a green value, a blue value.

So all the information can be encoded in three 2D arrays.
2d Arrays in Python

\[
A = \begin{bmatrix}
12 & 17 & 49 & 61 \\
38 & 18 & 82 & 77 \\
83 & 53 & 12 & 10 \\
\end{bmatrix}
\]

A list of lists.
# Accessing Entries

$$A = \begin{bmatrix} 12, 17, & 49, & 61 \\ 38, & 18, & 82, & 77 \\ 83, & 53, & 12, & 10 \end{bmatrix}$$

- $A[1][2] = 82$
### Accessing Entries

Let's consider the matrix:

$$
A = \begin{bmatrix}
12 & 17 & 49 & 61 \\
38 & 18 & 82 & 77 \\
83 & 53 & 12 & 10 \\
\end{bmatrix}
$$

To access the entry at row 2 and column 1, we would use $A[2][1]$.

$$
A[2][1] = 18
$$
Setting Up 2D Arrays

Here is a function that returns a reference to an m-by-n array of zeros:

```python
def zeros(m, n):
    v = []
    for k in range(n):
        v.append(0.0)
    A = []
    for k in range(m):
        A.append(v)
    return A
```
Python is Awkward

Turns out that base Python is not very handy for 2D array manipulations.

The `numpy` module makes up for this.

We will learn just enough `numpy` so that we can do elementary plotting, image processing and other things.
Introduction to 2D Arrays in numpy

A few essentials illustrated by examples.
Setting up a 2D Array of 0’s

```python
>>> from numpy import *
>>> m = 3
>>> n = 4
>>> A = zeros((m,n))
>>> A
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Note how the row and column dimensions are passed to `zeros`
Accessing an Entry

```python
>>> A = zeros((3,2))
>>> A[2,1] = 10
```

A nicer notation than A[2][1].
Accessing an Entry

>>> A = array([[1,2,3],[4,5,6]])

Using the array constructor to build a 3-by-2 array. Note all the square brackets.
Use Copy to Avoid Aliasing

```python
>>> A = array([[1,2],[3,4]])
>>> B = A
>>> A[1,1] = 10
>>> B
array([[ 1,  2],
        [ 3, 10]])
```

2D arrays are objects

```python
>>> A = array([[1,2],[3,4]])
>>> B = copy(A)
>>> A[1,1] = 10
>>> B
array([[ 1,  2],
        [ 3,  4]])
```
Iteration and 2D Arrays

Lots of Nested Loops
Nested Loops and 2D Arrays

A = array((3,3))
for i in range(3):
    for j in range(3):
        A[i,j] = (i+1)*(j+1)

A 3x3 times table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>
Nested Loops and 2D Arrays

\[ A = \text{array}((3,3)) \]

Allocates memory, but doesn’t put any values in the boxes. Much more efficient than the Repeated append framework.
for i in range(3):
    for j in range(3):
        A[i, j] = (i+1)*(j+1)

for i in range(3):
    A[i, 0] = (i+1)*(0+1)
    A[i, 1] = (i+1)*(1+1)
    A[i, 2] = (i+1)*(2+1)

Equivalent!
for $i$ in range(3):
    $A[i,0] = (i+1)*(0+1)$
    $A[i,1] = (i+1)*(1+1)$
    $A[i,2] = (i+1)*(2+1)$

Row 0 is set up when $i = 0$
Understanding 2D Array Set-Up

```python
for i in range(3):
    A[i,0] = (i+1)*(0+1)
    A[i,1] = (i+1)*(1+1)
    A[i,2] = (i+1)*(2+1)
```

Row 1 is set up when $i = 1$
Understanding 2D Array Set-Up

for i in range(3):
    A[i,0] = (i+1)*(0+1)
    A[i,1] = (i+1)*(1+1)
    A[i,2] = (i+1)*(2+1)

Row 2 is set up when i = 2
Extended Example

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.
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.
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 $j$. 

<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>

It costs $12 for factory 1 to make product 3.
The value of $I[k, j]$ is the inventory in factory $k$ of product $j$. 

\[
\begin{array}{cccccc}
38 & 5 & 99 & 34 & 42 \\
82 & 19 & 83 & 12 & 42 \\
51 & 29 & 21 & 56 & 87 \\
\end{array}
\]
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.
The value of $PO[j]$ is the number product $j$'s that the customer wants.
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:

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

Suppose:

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

Useful in functions and methods with 2D array arguments

\[(m,n) \text{ is a "tuple"}\]

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

\[m: 2 \quad n: 3\]

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.
The Class Company: Attributes

class Company(object):
    ""

    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
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

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]
Computing Total Value

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

Inventory Array

Cost Array
Computing Total Value

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

Inventory Array:

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

Cost Array:

\[
\begin{array}{ccc}
30 & 40 & 50 \\
60 & 70 & 80
\end{array}
\]
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

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

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

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

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

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

For factory 0:
For factory 0:

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

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

$$s = 0$$

for $j$ in range(5):
    $$s = += C[k,j] \times PO[j]$$
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

<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>
<tr>
<td>1</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

self.C -->

<table>
<thead>
<tr>
<th></th>
<th>1019</th>
</tr>
</thead>
<tbody>
<tr>
<td>930</td>
<td></td>
</tr>
<tr>
<td>1040</td>
<td></td>
</tr>
</tbody>
</table>

PO -->

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)?

<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></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>I</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>82</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>83</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>42</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>38</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>99</td>
<td>83</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
<td>87</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>

all(I[0,:] >= PO) is True
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[1, :] >= PO ) is False
Who Can Fill the Purchase Order (PO)?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>12</th>
<th>29</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>38</td>
<td>5</td>
<td>99</td>
<td>34</td>
<td>42</td>
</tr>
<tr>
<td>PO</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>

\text{all( I[2,:] \geq 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)
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)

Initial ize Who to the empty list. Then build it up thru repeated appending
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.
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 is not a numpy array, but array(Who) is
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

If \( k \) in kVals is True, then

\[
\text{all}(A.I[k, :]) \geq PO \quad \text{is True}
\]
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.
    """
    theCosts = Order(PO)
    Who = CanDo(PO)
    if len(Who) == 0:
        return None
    else:
```

PreC: PO is a purchase order list. """
Who Can Fill the Purchase Order Most Cheaply?

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>5</td>
<td>99</td>
<td>34</td>
<td>42</td>
<td></td>
</tr>
<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>

<table>
<thead>
<tr>
<th></th>
<th>PO</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

\[ k_{\text{Min}} = 0, \quad \text{cost}_{\text{Min}} = 1019 \]
Implementation

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

```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

The factory with index \( k_{\text{Min}} \) can deliver \( PO \) most cheaply and the cost is \( \text{cost}_{\text{Min}} \)

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

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

#### Before

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>5</td>
<td>99</td>
<td>34</td>
<td>42</td>
<td></td>
</tr>
<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>

#### After

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>12</td>
<td>29</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**PO -->** 1, 0, 12, 29, 5

**I -->** 82, 19, 83, 12, 42

**Yes** 1019

**No**

**Yes** 1040

**Before**
# Updating Inventory

<table>
<thead>
<tr>
<th>I</th>
<th>82</th>
<th>19</th>
<th>83</th>
<th>12</th>
<th>42</th>
</tr>
</thead>
<tbody>
<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  |

After
Method for Updating the Inventory Array After Processing a PO

```python
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]*P0[j]
```

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