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

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

C - - - >

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

Cost Array

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

C - - - - >

It costs $12 for factory 1 to make product 3

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

Inventory Array

38	5	99	34	42
82	19	83	12	42
51	29	21	56	87

I --->

The value of `I[k,j]` is the inventory in factory k of product j.

Inventory Array

38	5	99	34	42
82	19	83	12	42
51	29	21	56	87

I --->

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

The value of `I[k,j]` is the inventory in factory k of product j.

Purchase Order

PO ---> | 1 | 0 | 12 | 29 | 5 |
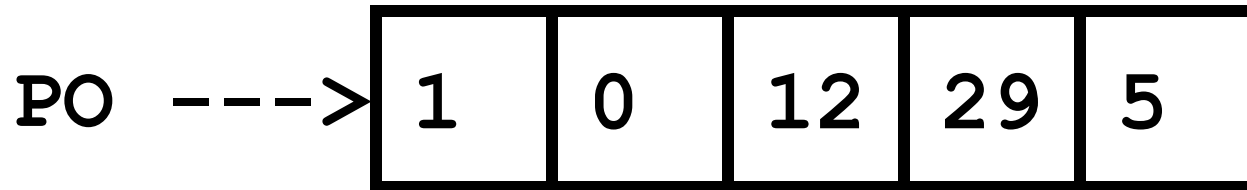
The value of `PO[j]` is the number
product j's that the customer wants

Purchase Order

PO ---->

1	0	12	29	5
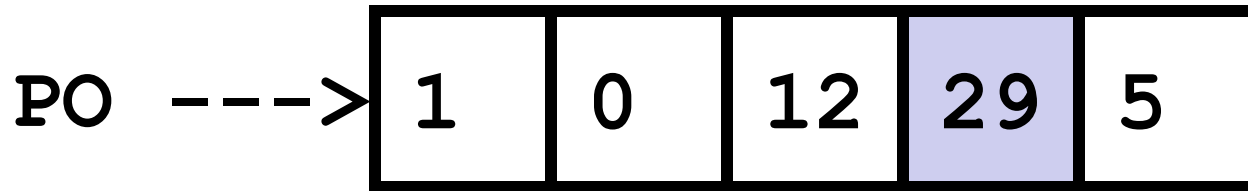
The customer wishes to purchase 29 product 3 units

The value of `PO[j]` is the number product j's that the customer wants

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:

10	36	22
12	35	20

I -->

A 2-by-3 array.

```
I = array([[10,36,22],[12,35,20]])
```

Computing Row and Column Dimension Using shape

Suppose:

10	36	22
12	35	20

I -->

Useful in functions and methods with 2D array arguments

(m,n) is a "tuple"

`(m,n) = I.shape`

m: **2**   n: **3**

shape is an attribute of the array class

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
```

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.

The Class Company: Attributes

```
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

```python
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]
```

The nested loop takes us to each array entry

10	36	22
12	35	20

I -->

30	40	50
60	70	80

C -->

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

10	36	22
12	35	20

I -->

30	40	50
60	70	80

C -->

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]
```

10	36	22
12	35	20

I -->

30	40	50
60	70	80

C -->

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

Inventory Array

C -->

30	40	50
60	70	80

Cost Array

Computing Total Value

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

10	36	22
12	35	20

I -->

Inventory Array

30	40	50
60	70	80

C -->

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

Inventory Array

C -->

30	40	50
60	70	80

Cost 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?

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

C --->

PO --->

1	0	12	29	5

For factory 0:

1*10 + 0*36 + 12*22 + 29* 15 + 5*62

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

C --->

1	0	12	29	5

PO --->

For factory 0:

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

10	**36**	22	15	62
12	35	20	12	66
13	37	21	16	59

C --->

j = 1

PO --->

1	**0**	12	29	5

For factory 0:

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

10	36	**22**	15	62
12	35	20	12	66
13	37	21	16	59

C --->

j = 2

1	0	**12**	29	5

PO --->

For factory 0:

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

10	36	22	**15**	62
12	35	20	12	66
13	37	21	16	59

C --->

j = 3

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

For factory 0:

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

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

C --->

j = 4

PO ---> | 1 | 0 | 12 | 29 | 5 |

For factory 0:

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

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

C --->

1	0	12	29	5

PO --->

For factory 1:

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

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

C --->

1	0	12	29	5

PO --->

For factory k:

```
s = 0;
for j in range(5):
    s = += C[k,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.
    """
```

What the Order Method Does

10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

self.C --> (points to second row)

1019

930

1040

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

```
>>> 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)?

38	5	99	34	42	**Yes**
82	19	83	12	42	**No**
51	29	21	56	87	**Yes**

I --->

PO --->

1	0	12	29	5

Factory 2 can't because 12 < 29

Who Can Fill the Purchase Order (PO)?

38	5	99	34	42	**Yes**
82	19	83	12	42	**No**
51	29	21	56	87	**Yes**

I -->

PO --> | 1 | 0 | 12 | 29 | 5 |

We need to compare the rows of I with PO.

Who Can Fill the Purchase Order (PO)?

38	5	99	34	42	Yes
I --> 82	19	83	12	42	No
51	29	21	56	87	Yes

PO --> | 1 | 0 | 12 | 29 | 5 |

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

Who Can Fill the Purchase Order (PO)?

38	5	99	34	42	**Yes**
82	**19**	**83**	**12**	**42**	**No**
51	29	21	56	87	**Yes**

I --> (points to row: 82 19 83 12 42)

PO --> | 1 | 0 | 12 | 29 | 5 |

```
all( I[1,:] >= PO )  is False
```

Who Can Fill the Purchase Order (PO)?

38	5	99	34	42	**Yes**
82	19	83	12	42	**No**
51	29	21	56	87	**Yes**

I -->

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)
```

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

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

Who Can Fill the PO?

```python
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?

```python
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

```
>>> 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??

For Q3 We Have

```
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?

38	5	99	34	42	Yes	1019
82	19	83	12	42	No	
51	29	21	56	87	Yes	1040

I -->

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

```
>>> 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

Updating Inventory

	38	5	99	34	42	Yes 1019
I -->	82	19	83	12	42	No
	51	29	21	56	87	Yes 1040

PO --> | 1 | 0 | 12 | 29 | 5 |

Before

Updating Inventory

37	5	87	5	37
82	19	83	12	42
51	29	21	56	87

I -->

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
        self.I[k,j] = self.I[k,j] - PO[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.