27. Two-Dimensional Arrays

Topics

- Motivation
- The numpy Module
- Subscripting
- functions and 2d Arrays

Visualizing

12	17	49	61
38	18	82	77
83	53	12	10

Can have a 2d array of strings or objects.

But we will just deal with 2d arrays of numbers.

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)

Rows and Columns

12	17	49	61
38	18	82	77
83	53	12	10

This is row 1.

Rows and Columns

12	17	49	61
38	18	82	77
83	53	12	10

This is column 2.

Entries

12	17	49	61
38	18	82	77
83	53	12	10

This is the (1,2) entry.

Where Do They Come From?

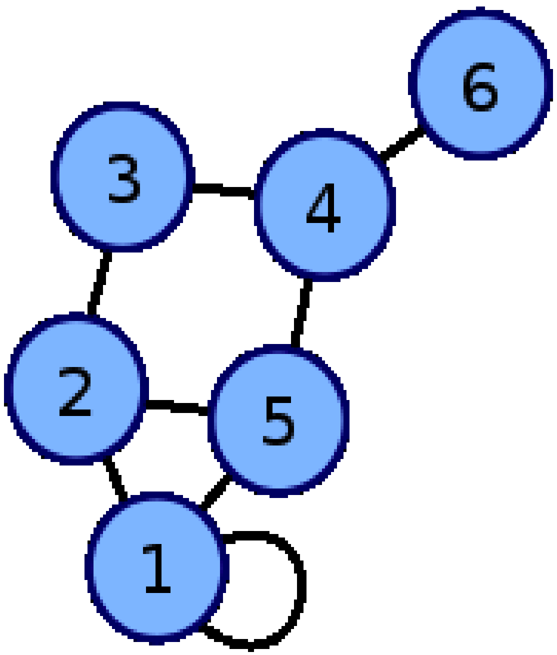
Entry (i,j) is the distance from city i to city j

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		Amsterdam	Berlin	Bordeaux	Brussels	Copenhagen	Dublin	Lisbon	London	Madrid	Milan	Munich	Paris	Rome	Zurich
2	Amsterdam	0	650.594	1084.367	204.7	766.456	946.404	2254.519	476.014	1783.664	1071.746	820.188	503.852	1657.55	818.784
3	Berlin	651.304	0	1634.132	764.787	379.95	1506.491	2804.284	1036.101	2333.429	1033.586	582.566	1053.617	1513.741	844.044
4	Bordeaux	1084.547	1630.51	0	890.135	1785.177	1444.887	1174.092	975.717	703.237	1018.437	1284.774	582.938	1508.036	1021.859
5	Brussels	207.37	767.381	891.025	0	908.03	775.414	2061.177	306.244	1590.322	881.246	784.539	310.51	1467.05	628.274
6	Copenhagen	768.376	381.155	1785.864	906.197	0	1646.681	2956.016	1177.511	2485.161	1414.722	1080.551	1205.349	2011.726	1185.589
7	Dublin	939.78	1499.75	1439.475	769.049	1640.41	0	2609.627	453.606	2138.772	1641.326	1554.938	863.552	2227.14	1388.364
8	Lisbon	2251.111	2797.07	1171.514	2056.699	2951.741	2611.451	0	2142.281	626.064	2150.158	2448.668	1749.502	2535.253	2185.753
9	London	478.973	1038.94	978.668	308.242	1179.603	455.078	2148.82	0	1677.965	1180.519	1094.131	402.745	1766.323	927.557
10	Madrid	1782.485	2328.44	702.888	1588.073	2483.115	2144.045	625.192	1673.655	0	1581.588	1978.157	1280.876	1966.683	1669.123
11	Milan	1074.297	1035.63	1019.438	905.951	1415.052	1672.432	2152.653	1202.042	1580.336	0	492.726	847.819	584.634	279.263
12	Munich	822.285	582.946	1282.395	783.498	1078.905	1559.472	2450.087	1090.302	1976.382	490.983	0	828.256	929.685	314.143
13	Paris	502.799	1048.75	583.225	308.387	1203.429	869.622	1753.377	400.452	1282.522	848.469	830.414	0	1418.908	653.608
14	Rome	1660.357	1514.24	1509.825	1492.011	1976.829	2257.272	2540.524	1788.102	1968.207	586.94	930.682	1431.299	0	865.323
15	Zurich	821.854	845.704	1021.829	653.218	1186.023	1419.699	2189.521	949.309	1668.309	279.652	315.164	653.299	865.456	0
16															

Distances / Times

Where Do they Come From?

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



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & \boxed{1} \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \boxed{1} & 0 & 0 \end{pmatrix}$$

Nodes 4 and 6 Are connected

Captures the connectivity in a network

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

12	17	49	61
38	18	82	77
83	53	12	10

```
A = [[12,17,49,61],[38,18,82,77],[83,53,12,10]]
```

A list of lists.

Accessing Entries

12	17	49	61
38	18	82	77
83	53	12	10

`A[1][2]`

```
A = [[12,17,49,61],[38,18,82,77],[83,53,12,10]]
```

Accessing Entries

12	17	49	61
38	18	82	77
83	53	12	10

`A[2][1]`

```
A = [[12,17,49,61],[38,18,82,77],[83,53,12,10]]
```

Setting Up 2D Arrays

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

```
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

```
>>> 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

```
>>> A = zeros((3,2))
>>> A[2,1] = 10
>>> A
array([[  0.,   0.],
       [  0.,   0.],
       [  0.,  10.]])
```

A nicer notation than A[2][1].

Accessing an Entry

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

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

1	2
3	4

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

2D arrays are objects

Iteration and 2D Arrays

Lots of Nested Loops

Nested Loops and 2D Arrays

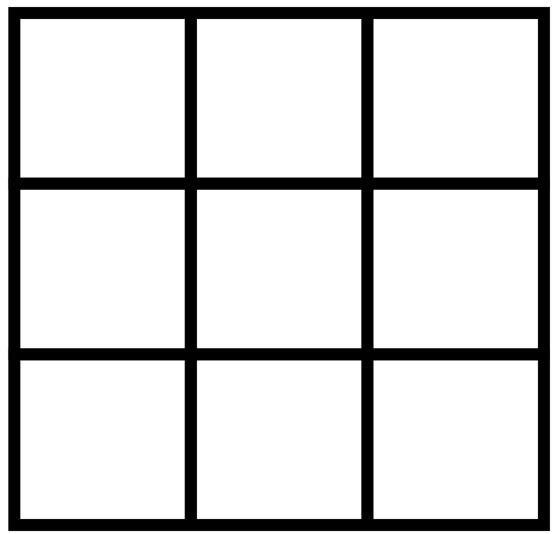
```
A = array((3,3))
for i in range(3):
    for j in range(3):
        A[i,j] = (i+1)*(j+1)
```

1	2	3
2	4	6
3	6	9

A
3x3
times
table

Nested Loops and 2D Arrays

```
A = array((3,3))
```



Allocates memory, but doesn't put any values in the boxes. Much more efficient than the Repeated append framework.

Understanding 2D Array Set-Up

```
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!

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)
```

1	2	3

Row 0 is set up when i = 0

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)
```

1	2	3
2	4	6

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)
```

1	2	3
2	4	6
4	6	9

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.

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

C ---->

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

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

Cost Array

C --->

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

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

I --->

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

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

Inventory Array

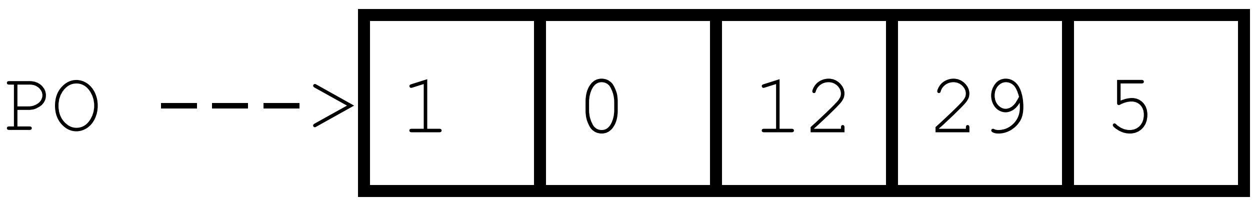
I --->

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

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

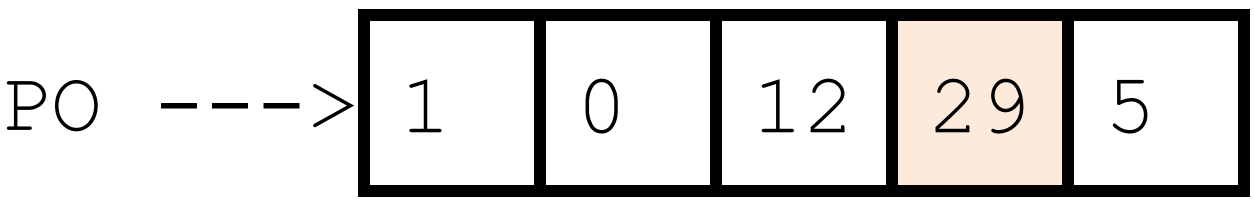
The value of `I[k,j]` is the inventory in factory k of product j.

Purchase Order



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

Purchase Order



The customer wishes to purchase 29 product 3 units

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

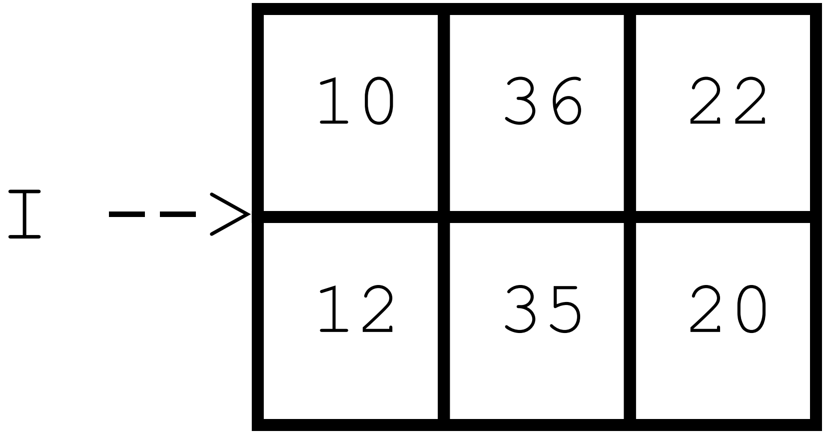
We Will Develop a Class called **`Company`**

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:



A 2-by-3 array.

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

Computing Row and Column Dimension Using shape

Suppose:

I -->

10	36	22
12	35	20

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(object):
    """
   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]
```

The nested loop takes us to each array entry

I -->

10	36	22
12	35	20

Inventory Array

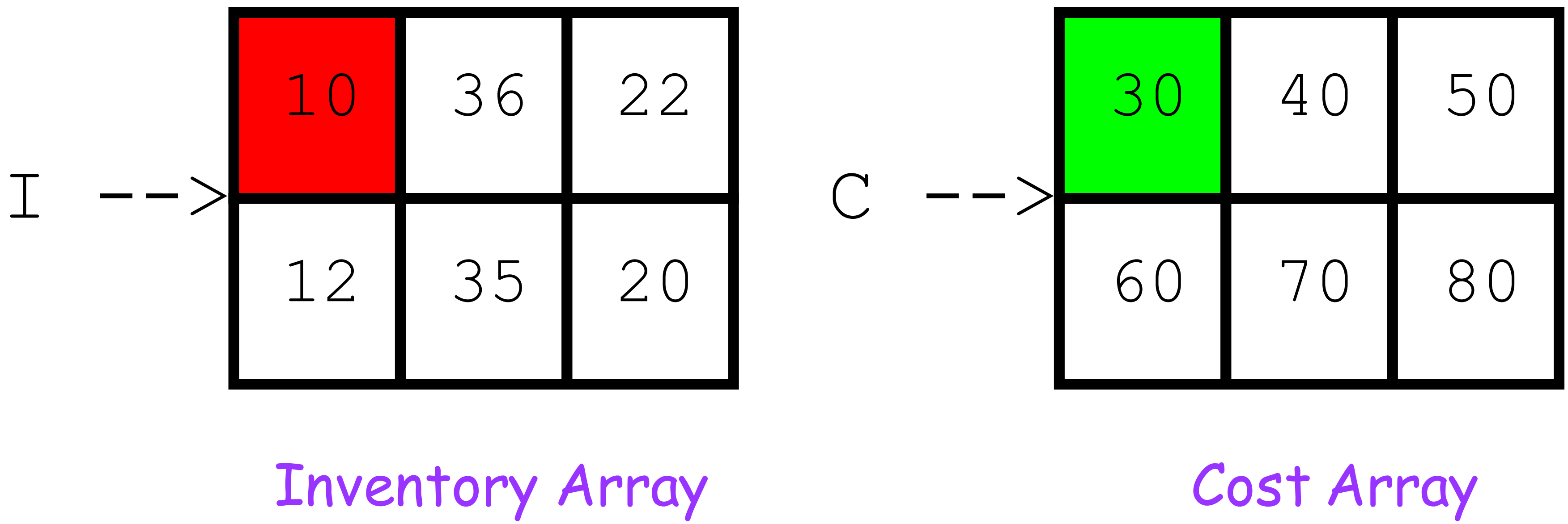
C -->

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]
```



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]
```

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]
```

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]
```

I -->

10	36	22
12	35	20

Inventory Array

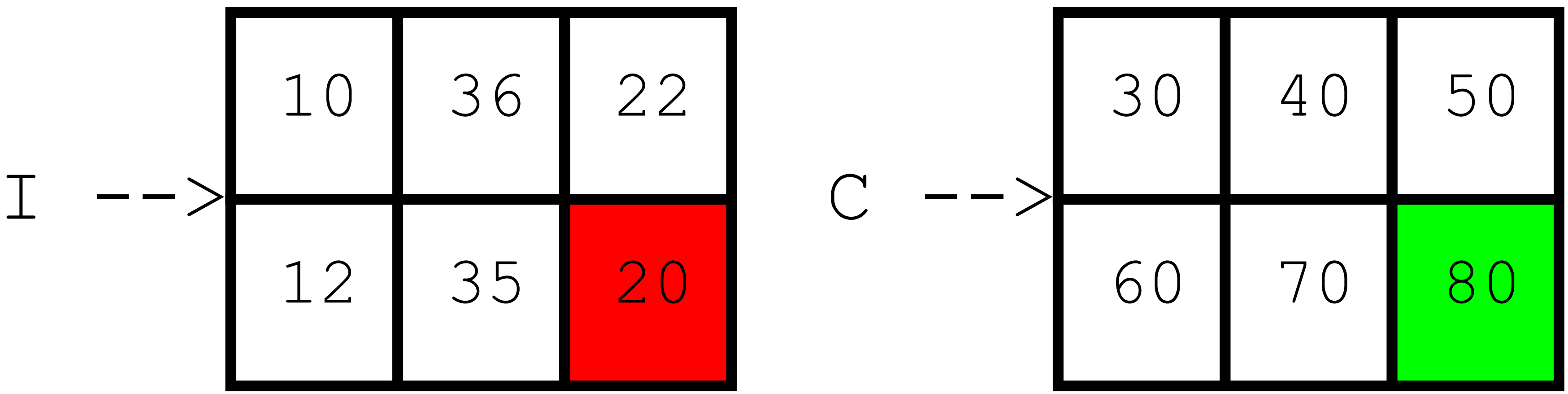
C -->

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]
```



Inventory Array   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?

C --->

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

PO --->

1	0	12	29	5

For factory 0:

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

C --->

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

j = 0

PO --->

1	0	12	29	5

For factory 0:

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

C --->

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

j = 1

PO --->

1	0	12	29	5

For factory 0:

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

C --->

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

j = 2

PO --->

1	0	12	29	5

For factory 0:

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

j = 3

C --->

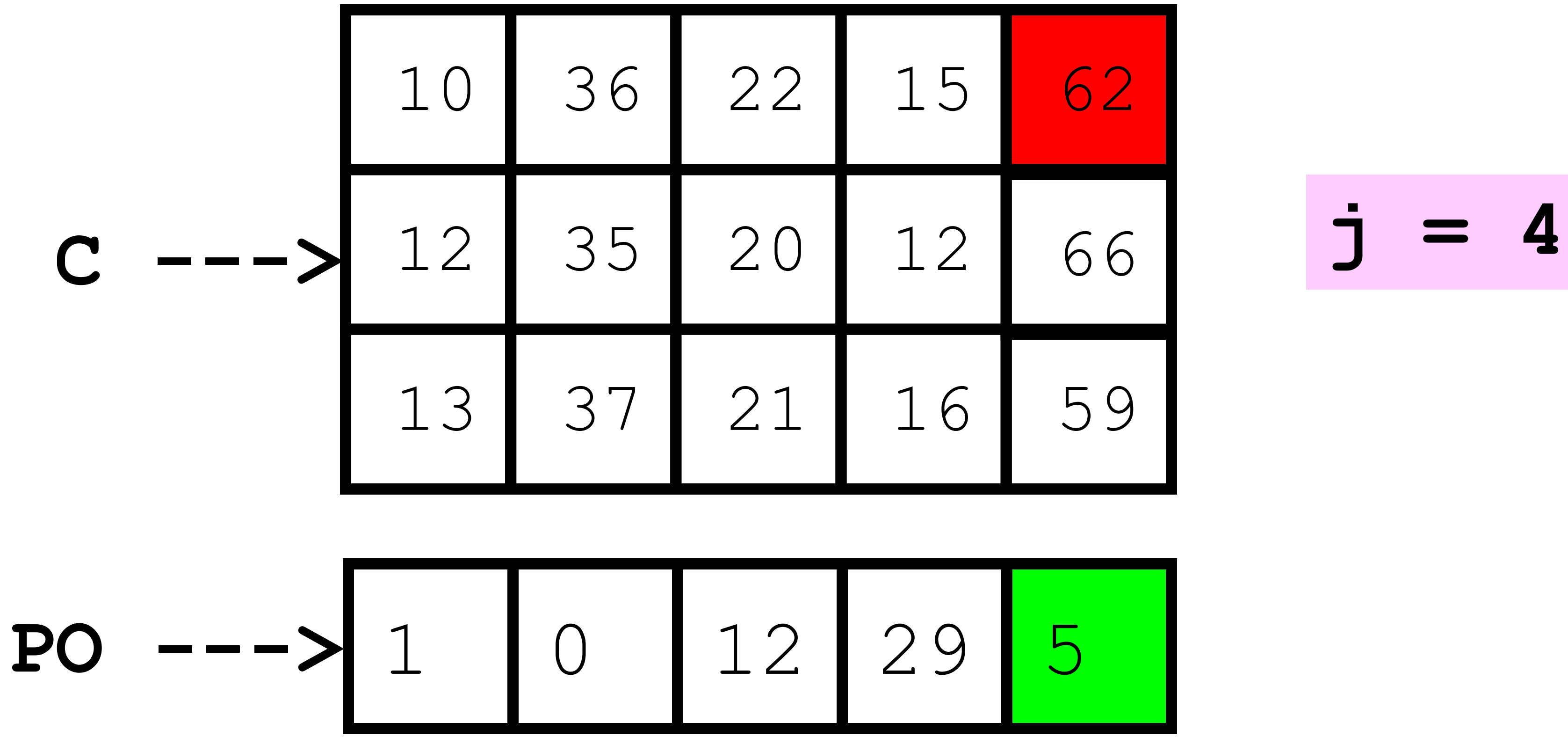
10	36	22	15	62
12	35	20	12	66
13	37	21	16	59

PO --->

1	0	12	29	5

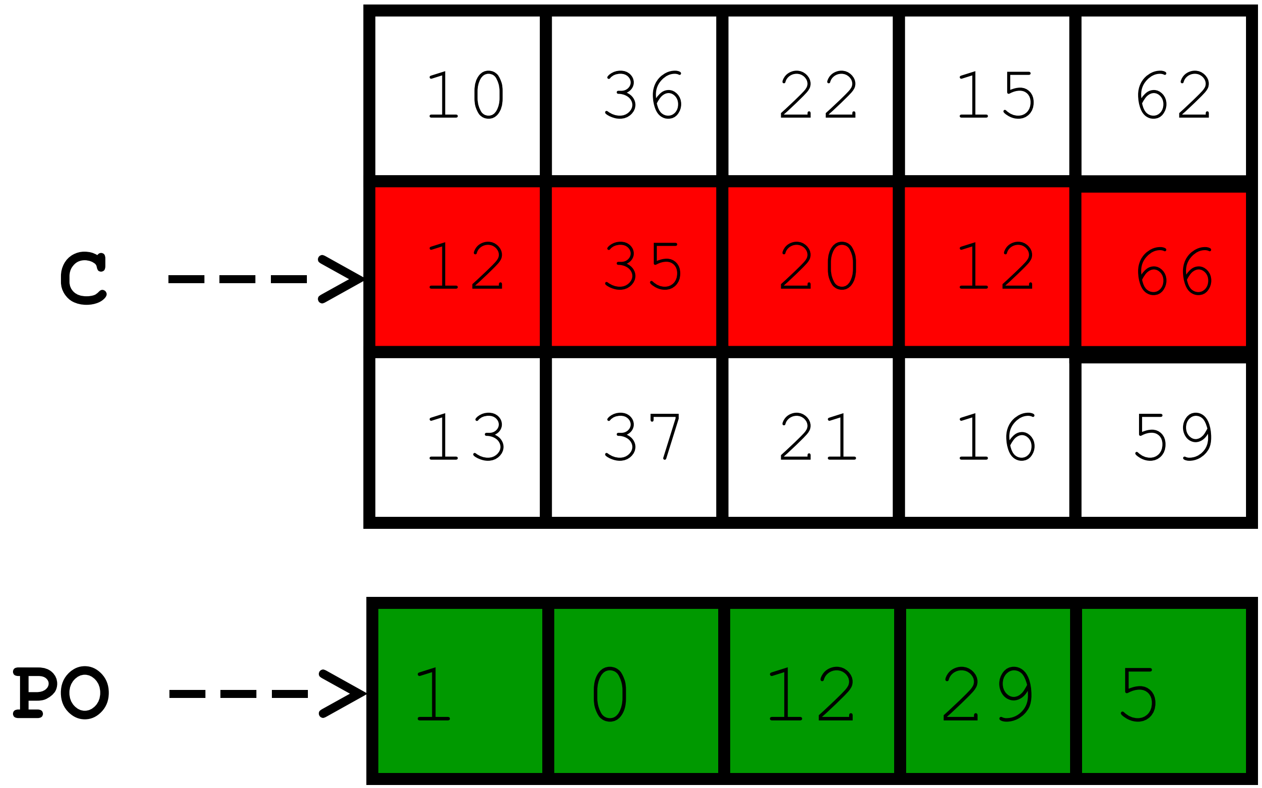
For factory 0:

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



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]
```

C --->

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

PO --->

1	0	12	29	5

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

self.C -->

10	36	22	15	62	**1019**
12	35	20	12	66	**930**
13	37	21	16	59	**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**
I -->	82	19	83	12	42	**No**
	51	29	21	56	87	**Yes**

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**
I -->	82	19	83	12	42	**No**
	51	29	21	56	87	**Yes**

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**
I -->	82	19	83	12	42	**No**
	51	29	21	56	87	**Yes**

PO -->	1	0	12	29	5

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

Who Can Fill the Purchase Order (PO)?

I -->

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

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?

```
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

```
>>> 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. """

         theCosts = Order(PO)
         Who = CanDo(P0)
         if len(Who)==0:
             return None
         else:
```

Who Can Fill the Purchase Order Most Cheaply?

	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

kMin = 0,   costMin = 1019

Implementation

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

Implementation Cont'd

```
# 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
I -->	82	19	83	12	42
	51	29	21	56	87

PO -->	1	0	12	29	5

After

Method for Updating the Inventory Array After Processing a PO

```
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.