25. Two-Dimensional Arrays

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

Visualizing

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

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

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

This is row 1.
This is column 2.

Entries

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

This is the (1,2) entry.

Where Do They Come From?

Entry (i,j) is the distance from city i to city j
Where Do they Come From?

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

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

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

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
array([[  0.,   0.],
       [  0.,   0.],
       [  0.,  10.]])
```

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

Accessing an Entry

```python
>>> 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

```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

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

<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

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

```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)
```

Equivalent!

### Understanding 2D Array Set-Up

For $i$ in range(3):

- $A[i,0] = (i+1)*(0+1)$
- $A[i,1] = (i+1)*(1+1)$
- $A[i,2] = (i+1)*(2+1)$

Row 0 is set up when $i = 0$

Row 1 is set up when $i = 1$
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 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

```
   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

\[
\begin{array}{cccc}
C & \rightarrow & 10 & 36 & 22 & 15 & 62 \\
   &        & 12 & 35 & 20 & 12 & 66 \\
   &        & 13 & 37 & 21 & 16 & 59 \\
\end{array}
\]

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

Inventory Array

\[
\begin{array}{cccc}
I & \rightarrow & 38 & 5 & 99 & 34 & 42 \\
   &        & 82 & 19 & 83 & 12 & 42 \\
   &        & 51 & 29 & 21 & 56 & 87 \\
\end{array}
\]

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

Inventory Array

\[
\begin{array}{cccc}
I & \rightarrow & 38 & 5 & 99 & 34 & 42 \\
   &        & 82 & 19 & 83 & 12 & 42 \\
   &        & 51 & 29 & 21 & 56 & 87 \\
\end{array}
\]

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

Purchase Order

\[
\begin{array}{cccc}
PO & \rightarrow & 1 & 0 & 12 & 29 & 5 \\
\end{array}
\]

The customer wishes to purchase 29 product 3 units.

Purchase Order

\[
\begin{array}{cccc}
PO & \rightarrow & 1 & 0 & 12 & 29 & 5 \\
\end{array}
\]

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:

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

A 2-by-3 array.

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

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

Inventory Array

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

Cost Array

<table>
<thead>
<tr>
<th>30</th>
<th>40</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>70</td>
<td>80</td>
</tr>
</tbody>
</table>

Computing Total Value

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

Inventory Array

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

Cost Array

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

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

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:

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

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

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

For factory 0:

$$s = 0;$$

for j in range(5):

$$s = += C[0,j] \times PO[j]$$

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

    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

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

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

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>Factory</th>
<th>Inventory</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38 5 99 34 42</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>82 19 83 12 42</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>51 29 21 56 87</td>
<td></td>
</tr>
<tr>
<td>Factory 2 can’t because 12 &lt; 29</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Who Can Fill the Purchase Order (PO)?

<table>
<thead>
<tr>
<th>Factory</th>
<th>Inventory</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38 5 99 34 42</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>82 19 83 12 42</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>51 29 21 56 87</td>
<td></td>
</tr>
<tr>
<td>Factory 2 can’t because 12 &lt; 29</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Who Can Fill the Purchase Order (PO)?

<table>
<thead>
<tr>
<th>Factory</th>
<th>Inventory</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38 5 99 34 42</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>82 19 83 12 42</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>51 29 21 56 87</td>
<td></td>
</tr>
<tr>
<td>all(I[0, :] &gt;= PO ) is True</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Who Can Fill the Purchase Order (PO)?

<table>
<thead>
<tr>
<th>Factory</th>
<th>Inventory</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38 5 99 34 42</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>82 19 83 12 42</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>51 29 21 56 87</td>
<td></td>
</tr>
<tr>
<td>all(I[1, :] &gt;= PO ) is False</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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[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.

Initialze Who to the empty list. Then build it up thru repeated appending.

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

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

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

If k in kVals is True, then

\[
\text{all}(A.I[k,:]>= PO)
\]

is True.

For Q3 We Have

def theCheapest(self,PO):
    
    """Return the tuple \((kMin,\text{costMin})\)
    where \(kMin\) is the index of the factory
    that can fill the PO most cheaply and
    \(\text{costMin}\) is the associated cost. If no
    such factory exists, return None.
    
    PreC: PO is a purchase order list. ""
    
    theCosts = Order(PO)
    Who = CanDo(PO)
    if len(Who)==0:
        return None
    else:
        # Find \(kMin\) and \(\text{costMin}\)
```

Who Can Fill the Purchase Order Most Cheaply?

<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>

\(kMin = 0, \text{costMin} = 1019\)

Implementation

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

Using Cheapest

Updating Inventory

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]*PO[j]
```

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