CS2204

Program Design and Data Structures for Scientific Computing

Announcements

- Project #2
  - We will be creating another class that allows us to represent & manipulate DNA strands
  - We will use a list as our underlying storage container
  - Our goal is to have a faster/smaller representation by utilizing the mutability of lists (as compared to the immutability of strings)
    - You should use mutation operations on your list whenever possible
    - Avoid operations that create new lists

Matrix Programming

Indexing

By Richard T. Guy

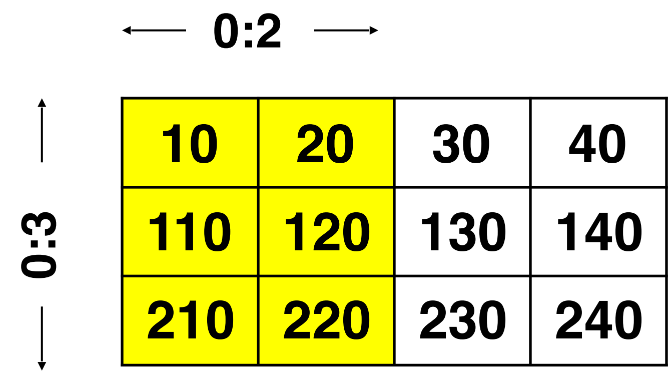
Now we'll have a look at some of the ways you can index arrays.

It may not seem important at first, but as we'll see, clever indexing allows you to avoid writing loops,

- reduces the size of your code
- makes it more efficient

Can *slice* arrays like lists or strings

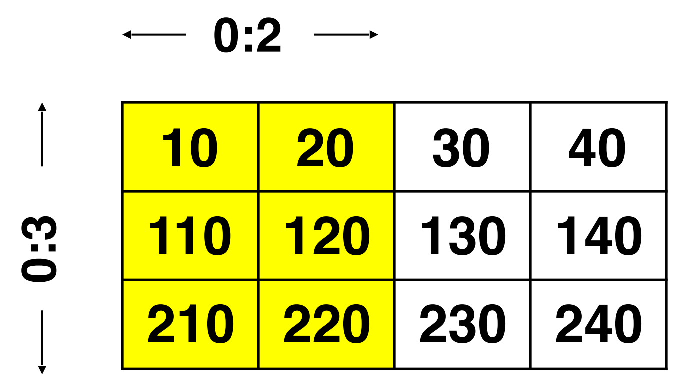
```
>>> block
array([[ 10,  20,  30,  40],
       [110, 120, 130, 140],
       [210, 220, 230, 240]])
>>> block[0:3, 0:2]
array([[ 10,  20],
       [110, 120],
       [210, 220]])
```



Can *slice* arrays like lists or strings

```
>>> block
array([[ 10,  20,  30,  40],
       [110, 120, 130, 140],
       [210, 220, 230, 240]])
>>> block[0:3, 0:2]
array([[ 10,  20],
       [110, 120],
       [210, 220]])
```

Are slices aliases or copies?



Can *slice* arrays like lists or strings

```
>>> block
array([[ 10,  20,  30,  40],
       [110, 120, 130, 140],
       [210, 220, 230, 240]])
>>> block[0:3, 0:2]
array([[ 10,  20],
       [110, 120],
       [210, 220]])
```

	0:2			
0:3	10	20	30	40
	110	120	130	140
	210	220	230	240

Are slices aliases or copies?

- *They are aliases*

Can assign to slices

```
>>> block[1, 1:3] = 0
>>> block
array([[ 10,  20,  30,  40],
       [110,   0,   0, 140],
       [210, 220, 230, 240]])
```

Slice on both sides to shift data

```
>>> vector = array([10, 20, 30, 40])
>>> vector[0:3] = vector[1:4]
>>> vector
array([20, 30, 40, 40])
```

Not overwritten

Slice on both sides to shift data

```
>>> vector = array([10, 20, 30, 40])
>>> vector[0:3] = vector[1:4]
>>> vector
array([20, 30, 40, 40])
```

Not overwritten

Is this easier to understand than a loop?

Slice on both sides to shift data

```
>>> vector = array([10, 20, 30, 40])
>>> vector[0:3] = vector[1:4]
>>> vector
array([20, 30, 40, 40])
```

Not overwritten

Is this easier to understand than a loop?

- In most cases, yes

Slice on both sides to shift data

```
>>> vector = array([10, 20, 30, 40])
>>> vector[0:3] = vector[1:4]
>>> vector
array([20, 30, 40, 40])
```

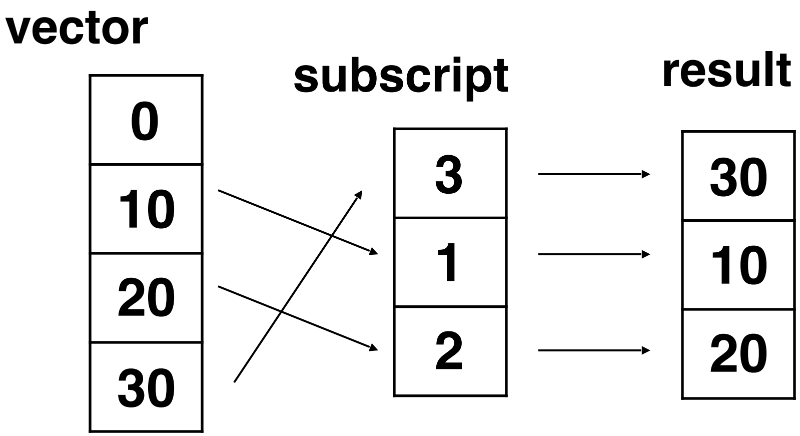
Not overwritten

Is this easier to understand than a loop?

- In most cases, yes
- Particularly when shifting up, when a loop would have to be written to count down

Can use lists or arrays as subscripts

```
>>> vector
array([0, 10, 20, 30])
>>> subscript = [3, 1, 2]
>>> vector[ subscript ]
array([30, 10, 20])
```



Also works in multiple dimensions

Though operation may not be obvious

```
>>> square = numpy.array([[5, 6], [7, 8]])
>>> square[ [1] ]
array([[7, 8]])
```

~~LOOPS~~

Comparisons

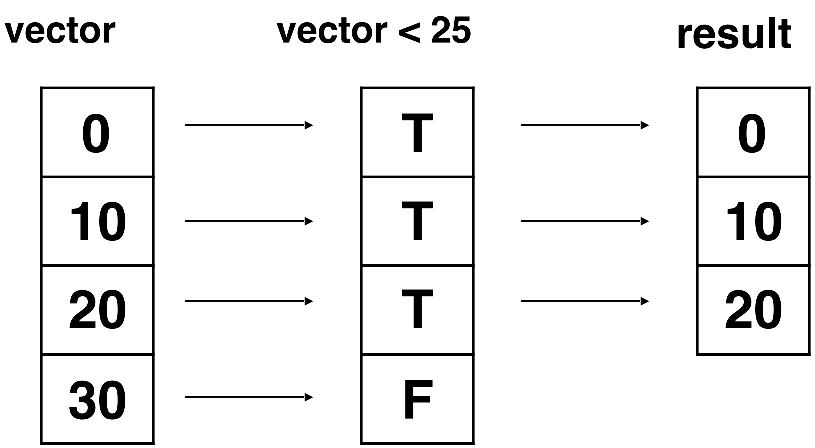
```
>>> vector
array([0, 10, 20, 30])
>>> vector < 25
array([ True, True, True, False ],
      dtype=bool)
```

← Data type is Boolean

The possible state values for a boolean object are True and False with the standard boolean operators, and, or, and not.

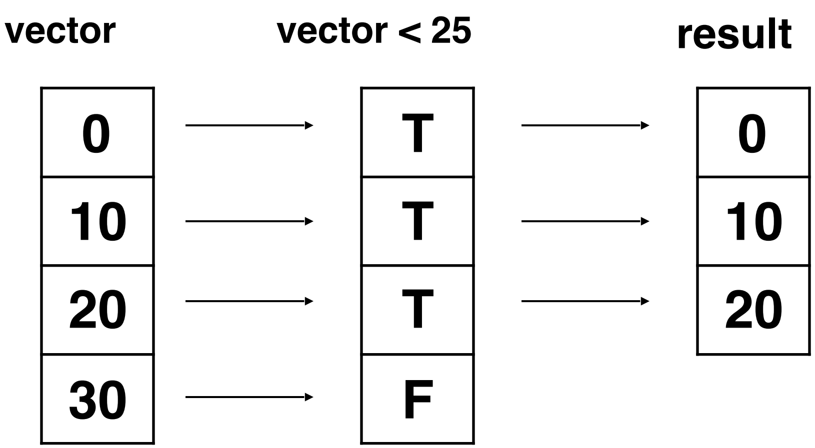
Use a Boolean subscript as a *mask*

```
>>> vector
array([0, 10, 20, 30])
>>> vector[ vector < 25 ]
array([0, 10, 20])
```



Use a Boolean subscript as a *mask*

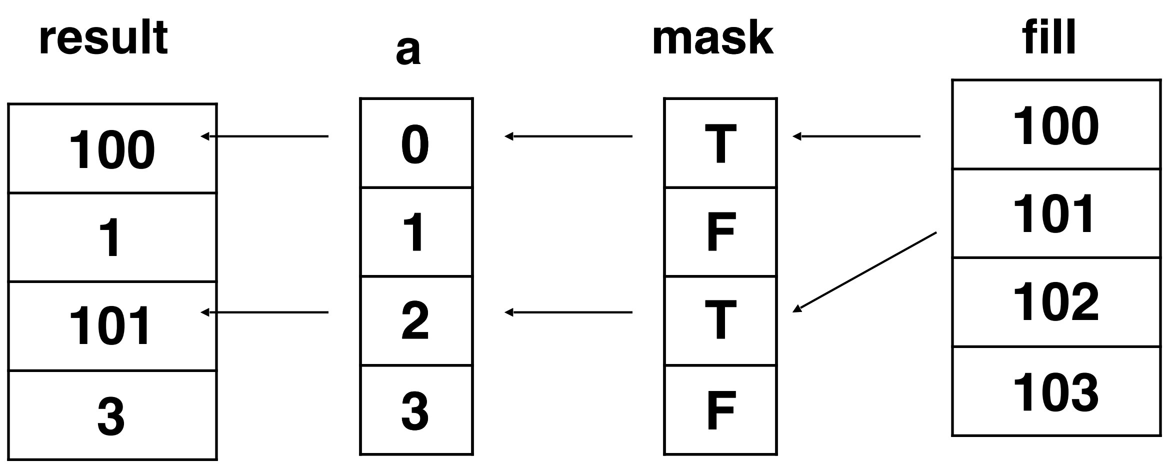
```
>>> vector
array([0, 10, 20, 30])
>>> vector[ vector < 25 ]
array([0, 10, 20])
```



Copy or alias?

Use masking for assignment

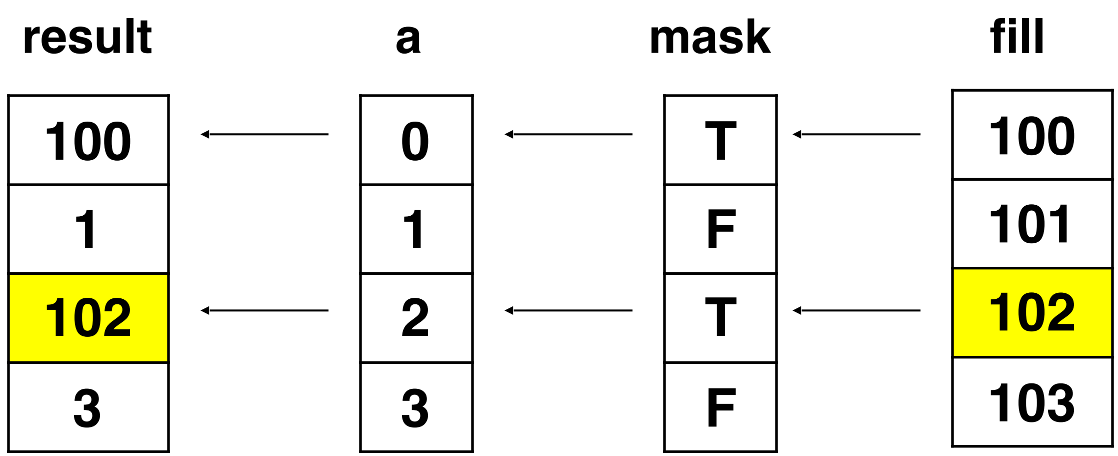
```
>>> a = array([0, 1, 2, 3])
>>> mask = array([True, False, True, False])
>>> a[mask] = array([100, 101, 102, 103])
>>> a
array([100, 1, 101, 3])
```



Taken in order

The **putmask** function works slightly differently

```
>>> a = array([0, 1, 2, 3])
>>> putmask(a, mask,
        array([100, 101, 102, 103]))
>>> a
array([100, 1, 102, 3])
```



Taken where True

Python does not allow objects to re-define the meaning of **and/or/not**

```
>>> vector = array([0, 10, 20, 30])
>>> vector <= 20
array([True, True, True, False], dtype=bool)
>>> (vector <= 20) and (vector >= 20)
ValueError: The truth value of an array with more than one element is ambiguous.
```

Use **logical_and** / **logical_or** functions

```
>>> logical_and(vector <= 20, vector >= 20)
array([False, False, True, False], dtype=bool)
```

Or use bitwise operations: **|** for **or**, **&** for **and**

```
>>> (vector <= 20) & (vector >= 20)
array([False, False, True, False], dtype=bool)
```

Use **where** instead of **if/else**

```
>>> vector = array([10, 20, 30, 40])
>>> where(vector < 25, vector, 0)
array([10, 20, 0, 0])
>>> where(vector > 25, vector/10, vector)
array([10, 20, 3, 4])
```

Review:

- Arrays can be sliced
- Or subscripted with vectors of indices
- Or masked with conditionals

Review:

– Arrays can be sliced

– Or subscripted with vectors of indices

– Or masked with conditionals

~~LOOPS~~

NumPy arrays make operations on rectangular data easy

But they are not quite mathematical matrices

```
>>> a = array([[1, 2], [3, 4]])
>>> a * a
```

Operators act *element-wise*

NumPy arrays make operations on rectangular data easy

But they are not quite mathematical matrices

```
>>> a = array([[1, 2], [3, 4]])
>>> a * a
    array([[ 1,  4],
           [ 9, 16]])
```

Operators act *element-wise*

So this does what you think

```
>>> a + a
```

And NumPy is sensible about *scalar* values

```
>>> a + 1
```

So this does what you think

```
>>> a + a
    array([[  2,  4],
           [  6,  8]])
```

And NumPy is sensible about *scalar* values

```
>>> a + 1
```

So this does what you think

```
>>> a + a
  array([[  2,  4],
         [  6,  8]])
```

And NumPy is sensible about *scalar* values

```
>>> a + 1
  array([[  2,  3],
         [  4,  5]])
```

Lots of useful utilities - Pay attention to dimensions!

>>> sum(a)

>>> sum(a, 0)

$$
\begin{array}{ccc}
 & 1 \rightarrow & \\
0 \downarrow & \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} & \begin{matrix}3 \\ 7\end{matrix} \\
 & \begin{matrix}4 & 6\end{matrix} &
\end{array}
$$

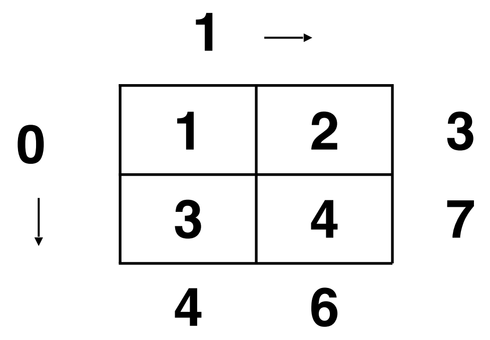
>>> sum(a, 1)

Lots of useful utilities - Pay attention to dimensions!

```
>>> sum(a)
    10

>>> sum(a, 0)

>>> sum(a, 1)
```



Lots of useful utilities - Pay attention to dimensions!

```
>>> sum(a)
    10

>>> sum(a, 0)
    array([4, 6])

>>> sum(a, 1)
```

$$\begin{array}{ccc c}
 & 1 \rightarrow & & \\
0 \downarrow & \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline\end{array} & & \begin{array}{c} 3 \\ 7 \end{array} \\
 & \begin{array}{cc} 4 & 6 \end{array} & &
\end{array}$$

Lots of useful utilities - Pay attention to dimensions!

```
>>> sum(a)
    10

>>> sum(a, 0)
    array([4, 6])

>>> sum(a, 1)
    array([3, 7])
```

$$
\begin{array}{cccc}
 & 1 & & \\
0 & \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline\end{array} & \begin{array}{c} 3 \\ 7 \end{array} \\
 & \begin{array}{cc} 4 & 6 \end{array} & &
\end{array}
$$

```
>>> data[:, 0]   # t0 count for all patients
array([1., 0., 0., 2., 1.])
>>> data[0, :]   # all samples for patient 0
array([1., 3., 3., 5., 12., 10., 9.])
```

Why are these 1D rather than 2D?

Example: Disease statistics

- We have 5 patients with 7 samples each
- One row per patient
- Columns are hourly responsive T cell counts

```
>>> data[:, 0]   # t0 count for all patients
array([1., 0., 0., 2., 1.])
>>> data[0, :]   # all samples for patient 0
array([1., 3., 3., 5., 12., 10., 9.])
```

Why are these 1D rather than 2D?

```
>>> mean(data)
6.8857
```

Gives us the average T cell count for all patients at all times

Intriguing, but not particularly meaningful

```
>>> mean(data, 0)   # over time
array([ 0.8, 2.6, 4.4, 6.4, 10.8, 11., 12.2])
```

The mean of the data along axis 0 gives us the average across all patients for each hour.

This is much more useful: it is the "normal" progress of the disease.

```
>>> mean(data)
6.8857
```

Gives us the average T cell count for all patients at all times

- Intriguing, but not particularly meaningful

```
>>> mean(data, 1)   # per patient
array([ 6.14, 4.28, 16.57, 2.14, 5.29])
```

The mean of the data along axis 1 gives us the average T cell count per patient across all times.

This could be useful if we need to normalize the data.

Select the data for people who started with a responsive T cell count of 0

```
>>> data[:, 0]
array([1., 0., 0., 2., 1.])
>>> data[:, 0] == 0.
array([False, True, True, False, False],
      dtype=bool)
>>> data[ data[:, 0] == 0 ]
array([[ 0., 1.,  2.,  4.,  8.,  7.,  8.],
       [ 0., 4., 11., 15., 21., 28., 37.]])
```

Find the mean T cell count over time for people who started with a count of 0

```
>>>    data[:, 0]
```

Column 0

Find the mean T cell count over time for people who started with a count of 0

```
>>> data[:, 0] == 0
```

Column 0 is 0

Find the mean T cell count over time for people who started with a count of 0

```
>>> data[ data[:, 0] == 0 ]
```

Rows where column 0 is 0

Find the mean T cell count over time for people who started with a count of 0

```
>>> mean(data[ data[:, 0] == 0 ], 0)
```

Mean along axis 0 of rows where column 0 is 0

Find the mean T cell count over time for people who started with a count of 0

```
>>> mean(data[ data[:, 0] == 0 ], 0)
array([ 0., 2.5, 6.5, 9.5, 14.5, 17.5, 22.5])
```

Find the mean T cell count over time for people who started with a count of 0

```
>>> mean(data[ data[:, 0] == 0 ], 0)
array([ 0., 2.5, 6.5, 9.5, 14.5, 17.5, 22.5])
```

Key to good array programming: no loops!

Just as true for MATLAB or R as for NumPy

What about "real" matrix multiplication?

```
>>> a = array([[1, 2], [3, 4]])
>>> dot(a, a)
```

```
>>> v = arange(3)   # [0, 1, 2]
>>> dot(v, v)       # 0*0 + 1*1 + 2*2
```

What about "real" matrix multiplication?

```
>>> a = array([[1, 2], [3, 4]])
>>> dot(a, a)
    array([[ 7, 10],
           [15, 22]])

>>> v = arange(3)   # [0, 1, 2]
>>> dot(v, v)       # 0*0 + 1*1 + 2*2
```

What about "real" matrix multiplication?

```
>>> a = array([[1, 2], [3, 4]])
>>> dot(a, a)
    array([[ 7, 10],
           [15, 22]])

>>> v = arange(3)   # [0, 1, 2]
>>> dot(v, v)       # 0*0 + 1*1 + 2*2
    5
```

Dot product only works for sensible shapes

```
>>> dot(ones((2, 3)), ones((2, 3)))
    ValueError: objects are not aligned
```

NumPy does not distinguish row/column vectors

```
>>> v = array([1, 2])
>>> a = array([[1, 2], [3, 4]])
>>> dot(v, a)
array([ 7, 10])
>>> dot(a, v)
array([ 5, 11])
```

$$\begin{bmatrix}1 & 2\end{bmatrix}\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = \begin{bmatrix}7\\ 10\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\begin{bmatrix}1\\ 2\end{bmatrix} = \begin{bmatrix}5 & 11\end{bmatrix}$$

Can also use the **matrix** subclass of **array**

```
>>> m = matrix([[1, 2], [3, 4]])
>>> m
matrix([[ 1, 2],
        [ 3, 4]])
>>> m*m  # '*' does matrix mult for matrices
matrix([[ 7, 10],
        [15, 22]])
```

Use **matrix(a)** or **array(m)** to convert

Which should you use?

If your problem is linear algebra, **matrix** will probably be more convenient

– Treats vectors as N×1 matrices

Otherwise, use **array**

– Especially if you're representing grids, rather than mathematical matrices

Always look at

http://www.scipy.org/Numpy_Example_List_With_Doc

before writing any functions of your own

conjugate **histogram**
convolve **lstsq**
correlate **npv**
diagonal **roots**
fft **solve**
gradient **svd**

Fast…

…and someone else has debugged them

I will also post an extended example with these lectures notes

Arrays vs. Lists

- Arrays and lists have many similarities, but there are also some important differences

- Similarities between arrays and lists:
  - Both are mutable: both can have elements reassigned in place
  - Arrays and lists are indexed and sliced identically
  - The len command works just as well on arrays as anything else
  - Arrays and lists both have sort and reverse attributes

- Differences between arrays and lists:
  - With arrays, the + and * signs do not refer to concatenation or repetition, but are element-wise addition/multiplication

Arrays vs. Lists

- The biggest difference between arrays and lists is speed; it's much faster to carry out operations on arrays (and all the terms therein) than on each term in a given list.
- Example: take the following script:

```
tt1 = time.clock()
sarr = 1.*arange(0,10001)/10000
sinarr = sin(sarr)
tt2 = time.clock()
slist = []; sinlist = []
for i in range(10001):
  slist.append(1.*i/10000)
  sinlist.append(sin(slist[i]))
tt3 = time.clock()
```

- `tt2-tt1` (the time to process with arrays) is 0.0007 seconds, while `tt3-tt2` (the time to process with lists) is 0.027 seconds.