



Python Programming

IOE 373 Lecture 17



Topics

- Tuples
- NumPy
 - Arrays
 - Array math



Tuples Are Like Lists

- Tuples are another kind of sequence that functions much like a list
 - They have elements which are indexed starting at 0
 - A tuple is a comma-separated list of values. (it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code)

```
>>> x = ('Glenn', 'Sally', 'Joseph')
```

```
>>> print(x[2])
```

```
Joseph
```

```
>>> y = 1, 9, 2
```

```
>>> print(y)
```

```
(1, 9, 2)
```

```
>>> print(max(y))
```

```
9
```

```
>>> for iter in y:  
...     print(iter)
```

```
...
```

```
1
```

```
9
```

```
2
```

```
>>>
```



Tuples are “immutable”

Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
>>> [9, 8, 6]
>>>
```

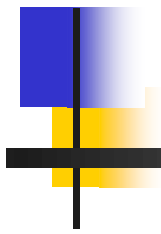
```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback: 'str'
object does
not support item
Assignment
>>>
```

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback: 'tuple'
object does
not support item
Assignment
>>>
```



Tuples are not lists

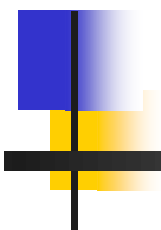
```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no attribute 'reverse'
>>>
```



Lists vs Tuples

```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

```
>>> t = tuple()
>>> dir(t)
['count', 'index']
```



Tuples are More Efficient

- Since tuple structures are not modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- If you need temporary variables, use tuples instead of lists, for more efficient and quicker scripts!



Tuples and Assignment

We can also put a tuple on the left-hand side of an assignment statement

We can even omit the parentheses

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred
>>> (a, b) = (99, 98)
>>> print(a)
99
```




Tuples and Dictionaries

The items() method
in dictionaries
returns a list of (key,
value) tuples

```
>>> d = dict()
>>> d['csev'] = 2
>>> d['cwen'] = 4
>>> for (k,v) in d.items():
...     print(k, v)
...
csev 2
cwen 4
>>> tups = d.items()
>>> print(tups)
dict_items([('csev', 2), ('cwen', 4)])
```



Tuples are Comparable

The comparison operators work with tuples and other sequences. If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```



Sorting Lists of Tuples

We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary

First we sort the dictionary by the key using the `items()` method and `sorted()` function

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> d.items()
dict_items([('a', 10), ('c', 22), ('b', 1)])
>>> sorted(d.items())
[('a', 10), ('b', 1), ('c', 22)]
```



Using sorted()

We can do this
even more
directly using
the built-in
function `sorted`
that takes a
sequence as a
parameter and
returns a sorted
sequence

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = sorted(d.items())
>>> t
[('a', 10), ('b', 1), ('c', 22)]
>>> for k, v in sorted(d.items()):
...     print(k, v)
...
a 10
b 1
c 22
```



Sort by Values Instead of Key

If we could construct a list of tuples of the form (value, key) we could sort by value

We do this with a for loop that creates a list of tuples

```
c = {'a':10, 'b':1, 'c':22}
```

```
tmp = list()
```

```
for k, v in c.items() :
```

```
    tmp.append( (v, k) )
```

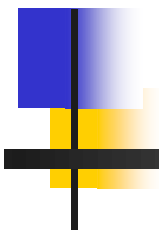
```
print(tmp)
```

```
>>[(10, 'a'), (22, 'c'), (1, 'b')]
```

```
tmp = sorted(tmp, reverse=True)
```

```
print(tmp)
```

```
>>[(22, 'c'), (10, 'a'), (1, 'b')]
```



```
fhand = open('romeo.txt')
counts = {}
for line in fhand:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0 ) + 1

lst = []
for key, val in counts.items():
    newtup = (val, key)
    lst.append(newtup)

lst = sorted(lst, reverse=True)

for val, key in lst[:10] :
    print(key, val)
```

**The top 10 most
common words**



Even Shorter Version

```
>>> c = {'a':10, 'b':1, 'c':22}

>>> print( sorted( [ (v,k) for k,v in c.items() ] ) )

[(1, 'b'), (10, 'a'), (22, 'c')]
```

List comprehension creates a dynamic list. In this case, we make a list of reversed tuples and then sort it.

<http://wiki.python.org/moin/HowTo/Sorting>



Numpy

- Numpy is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.



Arrays

- A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers.
- The number of dimensions is the rank of the array
- The shape of an array is a tuple of integers giving the size of the array along each dimension.



Arrays

- We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np
```

```
a = np.array([1, 2, 3])    # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"
```

```
b = np.array([[1,2,3],[4,5,6]]) # Create a rank 2 array
print(b.shape)                 # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 4"
```

Array Creation

More Examples

```
import numpy as np
```

```
a = np.zeros((2,2))      # Create an array of all zeros
print(a)                # Prints "[[ 0.  0.]
                        #      [ 0.  0.]]"
```

```
b = np.ones((1,2))      # Create an array of all ones
print(b)                # Prints "[[ 1.  1.]]"
```

```
c = np.full((2,2), 7)   # Create a constant array
print(c)                # Prints "[[ 7.  7.]
                        #      [ 7.  7.]]"
```

```
d = np.eye(2)           # Create a 2x2 identity matrix
print(d)                # Prints "[[ 1.  0.]
                        #      [ 0.  1.]]"
```

```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)                 # Might print "[[ 0.91940167  0.08143941]
                        #      [ 0.68744134  0.87236687]]"
```

Array Indexing

- **Slicing:** Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np
```

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

Create the following rank 2 array
with shape (3, 4)

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
b = a[:2, 1:3]
```

Use slicing to pull out the subarray
consisting of the first 2 rows
and columns 1 and 2; b is the
following array of shape (2, 2):

```
# [[2 3]
#  [6 7]]
```

```
# A slice of an array is a view into the same data, so modifying it  
# will modify the original array.
```

```
print(a[0, 1])    # Prints "2"  
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]  
print(a[0, 1])    # Prints "77"
```



Integer Array Indexing

```
import numpy as np
```

```
# Create a new array from which we will select elements
```

```
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
```

```
print(a)
```

```
# Create an array of indices
```

```
b = np.array([0, 2, 0, 1])
```

```
# prints "array([[ 1,  2,  3],  
#               [ 4,  5,  6],  
#               [ 7,  8,  9],  
#               [10, 11, 12]])"
```

```
# Select one element from each row of a using the indices in b
```

```
print(a[np.arange(4), b]) # Prints "[ 1  6  7 11]"
```

```
# Mutate one element from each row of a using the indices in b
```

```
a[np.arange(4), b] += 10
```

```
print(a) # prints "array([[11,  2,  3],
```

```
      #           [ 4,  5, 16],
```

```
      #           [17,  8,  9],
```

```
      #           [10, 21, 12]])
```



Boolean Indexing

```
import numpy as np
```

```
a = np.array([[1,2], [3, 4], [5, 6]])
```

```
bool_idx = (a > 2)  
print(bool_idx)
```

```
# Prints "[[False False]  
#         [ True  True]  
#         [ True  True]]"
```

```
# We use boolean array indexing to construct a rank 1 array  
# consisting of the elements of a corresponding to the True values of bool_idx
```

```
print(a[bool_idx]) # Prints "[3 4 5 6]"
```

```
# We can do all of the above in a single concise statement:  
print(a[a > 2]) # Prints "[3 4 5 6]"
```

Find the elements of a that are bigger than 2;
this returns a numpy array of Booleans of the
Same shape as a, where each slot of bool_idx tells
whether that element of a is > 2.



Array Datatypes

- Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

```
import numpy as np
```

```
x = np.array([1, 2])  
print(x.dtype)
```

```
# Let numpy choose the datatype  
# Prints "int64"
```

```
x = np.array([1.0, 2.0])  
print(x.dtype)
```

```
# Let numpy choose the datatype  
# Prints "float64"
```

```
x = np.array([1, 2], dtype=np.int64)  
print(x.dtype)
```

```
# Force a particular datatype  
# Prints "int64"
```

Array Math

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
print(x * y)
print(np.multiply(x, y))
```

```
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
```

```
print(x / y)
print(np.divide(x, y))
```

```
# Elementwise division; both produce the array
# [[ 0.2      0.33333333]
#  [ 0.42857143  0.5      ]]
```

```
print(x + y)
print(np.add(x, y))
```

```
# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
```

```
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
```

```
print(np.sqrt(x))
```

```
# Elementwise square root; produces the array
# [[ 1.      1.41421356]
#  [ 1.73205081  2.      ]]
```




Matrix/Vector Operations

- Use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]])  
y = np.array([[5,6],[7,8]])
```

```
v = np.array([9,10])  
w = np.array([11, 12])
```

```
# Inner product of vectors; both produce 219  
print(v.dot(w))  
print(np.dot(v, w))
```

```
# Matrix / vector product; both produce  
the rank 1 array [29 67]  
print(x.dot(v))  
print(np.dot(x, v))
```

```
# Matrix / matrix product; both produce  
the rank 2 array  
# [[19 22]  
# [43 50]]  
print(x.dot(y))  
print(np.dot(x, y))
```

Computations/Statistics on Arrays

- Full list: <https://numpy.org/doc/stable/reference/routines.math.html>

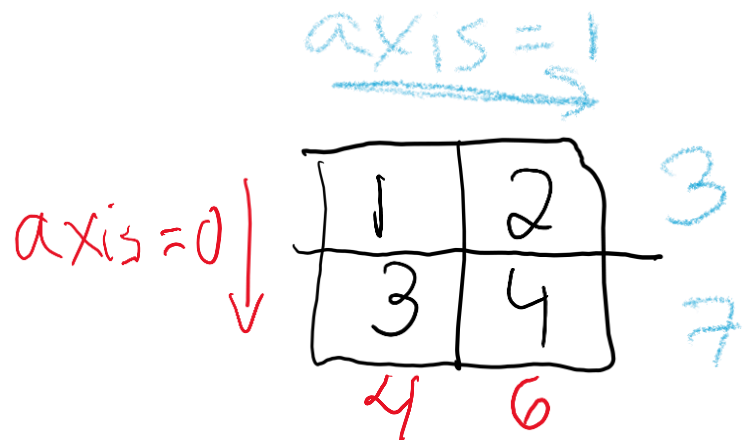
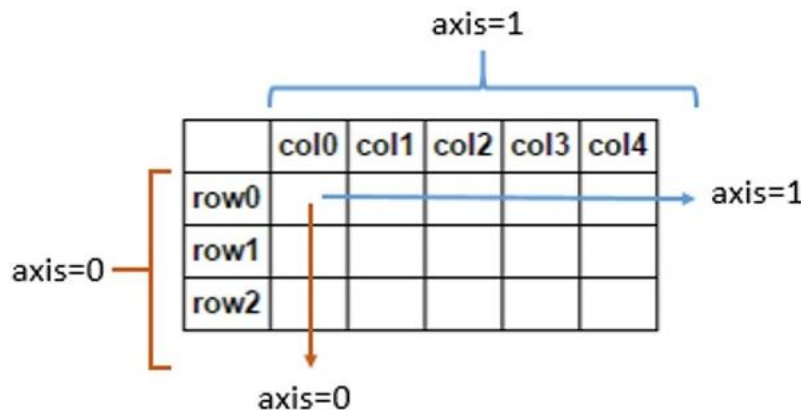
```
import numpy as np
```

```
x = np.array([[1,2],[3,4]])
```

```
print(np.sum(x)) # Compute sum of all elements; prints "10"
```

```
print(np.sum(x, axis=0)) # Compute sum of each column; prints "[4 6]"
```

```
print(np.sum(x, axis=1)) # Compute sum of each row; prints "[3 7]"
```





Array Transpose

```
import numpy as np
```

```
x = np.array([[1,2], [3,4]])  
print(x)                # Prints "[[1 2]  
                        #      [3 4]]"
```

```
print(x.T)               # Prints "[[1 3]  
                        #      [2 4]]"
```

Note that taking the transpose of a rank 1 array does nothing:

```
v = np.array([1,2,3])  
print(v)  # Prints "[1 2 3]"  
print(v.T) # Prints "[1 2 3]"
```



Array Broadcasting

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v # Add v to each row of x using broadcasting
print(y)      # Prints "[[ 2  2  4]
                #      [ 5  5  7]
                #      [ 8  8 10]
                #      [11 11 13]]"
```

The line `y = x + v` works even though `x` has shape (4, 3) and `v` has shape (3,) due to broadcasting; this line works as if `v` actually had shape (4, 3), where each row was a copy of `v`, and the sum was performed elementwise



Broadcasting Rules

- The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
- In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

<https://numpy.org/doc/stable/user/basics.broadcasting.html>