Core Tools I

Lecture 02

Kyle Bradbury and Leslie Collins

Outline

1. Python language essentials
2. Working with numerical data in arrays (Numpy and Pandas)
3. Plotting (Matplotlib and Seaborn)

The python/numpy/matplotlib portion of this lecture is from an excellent tutorial by Justin Johnson from Stanford, and the pandas portion of the tutorial was drawn from 10 Minutes to Pandas within the official pandas documentation.

Python

Data Types

- **Booleans**
- **Numbers** (integers, floats, long, and complex numbers).
- **Strings**
- Sequences
    - Bytes and byte arrays
    - **Lists** (ordered, mutable)
    - **Tuples** (ordered, immutable)
    - Sets (unordered, mutable)
- Mappings
    - Dictionaries

We'll discuss the **bolded** items in this tutorial

Python booleans

In [1]:
```
t = True
f = False
print(type(t))
```

```
<class 'bool'>
```

In [2]:
```
print(t and f) # Logical AND
```

```
False
```

In [3]:
```
print(t or f)  # Logical OR
```

```
True
```

In [4]:
```
print(not t)   # Logical NOT
```

```
False
```

In [5]:
```
print(t != f)  # Logical XOR
```

```
True
```

Python numbers

In [6]:
```
x = 3
print(type(x))
```

```
<class 'int'>
```

In [7]:
```
print(x)
```

```
3
```

In [8]:
```
print(x + 1)   # Addition
```

```
4
```

In [9]:
```
print(x - 1)   # Subtraction
```

```
2
```

In [10]:
```
print(x * 2)   # Multiplication
```

```
6
```

Python numbers (cont)

In [11]:
```
print(x ** 2)  # Exponentiation
```

```
9
```

In [12]:
```
x += 1
print(x)
```

```
4
```

In [13]:
```
x *= 2
print(x)
```

```
8
```

In [14]:
```
y = 2.5
print(type(y)) # Prints "<class 'float'>"
```

```
<class 'float'>
```

In [15]:
```
print(y, y + 1, y * 2, y ** 2)
```

```
2.5 3.5 5.0 6.25
```

Python strings

In [16]:
```
hello = 'hello'     # String literals can use single quotes
world = "world"     # or double quotes; it does not matter.
print(hello)
```

```
hello
```

In [17]:
```
print(len(hello))  # String length
```

```
5
```

In [18]:
```
hw = hello + ' ' + world  # String concatenation
print(hw)
```

```
hello world
```

In [19]:
```
hw12 = 'I said, {} {} = {}'.format(hello, world, 12)  # sprintf style string formatting
print(hw12)
```

```
I said, hello world = 12
```

Python strings (cont)

In [20]:
```
s = "hello"
print(s.capitalize())  # Capitalize a string
```

```
Hello
```

In [21]:
```
print(s.upper())       # Convert a string to uppercase
```

```
HELLO
```

In [22]:
```
print(s.rjust(7))      # Right-justify a string, padding with spaces
```

```
  hello
```

In [23]:
```
print(s.center(7))     # Center a string, padding with spaces
```

```
 hello
```

In [24]:
```
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another
```

```
he(ell)(ell)o
```

In [25]:
```
print('  world '.strip())  # Strip leading and trailing whitespace
```

```
world
```

Python sequences (data containers/structures)

Lists, dictionaries, tuples, sets, and strings. Lists and dictionaries are mutable (contents can change), the rest are immutable.

Python lists

In [26]:
```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])
```

```
[3, 1, 2] 2
```

In [27]:
```
print(xs[-1])     # Negative indices count from the end of the list
```

```
2
```

In [28]:
```
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)
```

```
[3, 1, 'foo']
```

In [29]:
```
xs.append('bar')  # Add a new element to the end of the list
print(xs)
```

```
[3, 1, 'foo', 'bar']
```

Python sequences: List slicing

In [30]:
```
nums = list(range(5))      # range is a built-in function that creates a list of integers
print(nums)
```

```
[0, 1, 2, 3, 4]
```

In [31]:
```
print(nums[2:4])           # Get a slice from index 2 to 4 (exclusive)
```

```
[2, 3]
```

In [32]:
```
print(nums[2:])            # Get a slice from index 2 to the end
```

```
[2, 3, 4]
```

In [33]:
```
print(nums[:2])            # Get a slice from the start to index 2 (exclusive)
```

```
[0, 1]
```

Python sequences: List slicing (cont)

In [34]:
```
print(nums[:])              # Get a slice of the whole list
```

```
[0, 1, 2, 3, 4]
```

In [35]:
```
print(nums[:-1])            # Slice indices can be negative
```

```
[0, 1, 2, 3]
```

In [36]:
```
nums[2:4] = [8, 9]          # Assign a new sublist to a slice
print(nums)
```

```
[0, 1, 8, 9, 4]
```

Python sequences: Loops

In [37]:
```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
```

```
cat
dog
monkey
```

Python sequences: Loops (continued)

In [38]:
```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

Python sequences: List comprehensions

Starting with this code...

In [39]:
```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)
```

```
[0, 1, 4, 9, 16]
```

...you can make it simpler with list comprehensions:

In [40]:
```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)
```

```
[0, 1, 4, 9, 16]
```

Python sequences: List comprehensions (cont)

...and you can add conditions:

In [41]:
```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

```
[0, 4, 16]
```

Python functions

In [42]:
```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

```
negative
zero
positive
```

Python functions: optional keyword arguments

In [43]:
```
def hello(name, loud=False):
    if loud:
        print('HELLO, {}!'.format(name.upper()))
    else:
        print('Hello, {}'.format(name))

hello('Bob')
hello('Fred', loud=True)
```

```
Hello, Bob
HELLO, FRED!
```

Python classes

In [44]:
```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, {}!'.format(self.name.upper()))
        else:
            print('Hello, {}'.format(self.name))

g = Greeter('Fred')  # Construct an instance of the Greeter class
```

In [45]:
```
g.greet()            # Call an instance method
```

```
Hello, Fred
```

In [46]:
```
g.greet(loud=True)   # Call an instance method
```

```
HELLO, FRED!
```

Working with numerical data in arrays: Numpy and Pandas

Numpy: Basics

In [47]:
```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))
```

```
<class 'numpy.ndarray'>
```

In [48]:
```
print(a.shape)
```

```
(3,)
```

In [49]:
```
print(a[0], a[1], a[2])
```

```
1 2 3
```

Numpy: Basics (cont)

In [50]:
```
a[0] = 5                  # Change an element of the array
print(a)
```

```
[5 2 3]
```

In [51]:
```
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b)
print(b.shape)
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

In [52]:
```
print(b[0, 0], b[0, 1], b[1, 0])
```

```
1 2 4
```

Numpy: Basic matrices

In [53]:
```
a = np.zeros((2,2))   # Create an array of all zeros
print(a)
```

```
[[ 0.  0.]
 [ 0.  0.]]
```

In [54]:
```
b = np.ones((1,2))    # Create an array of all ones
print(b)
```

```
[[ 1.  1.]]
```

In [55]:
```
c = np.full((2,2), 7)  # Create a constant array
print(c)
```

```
[[7 7]
 [7 7]]
```

Numpy: Basic matrices (cont)

In [56]:
```
d = np.eye(2)          # Create a 2x2 identity matrix
print(d)
```

```
[[ 1.  0.]
 [ 0.  1.]]
```

In [57]:
```
e = np.random.random((2,2))  # Create an array filled with random values
print(e)
```

```
[[ 0.78500112  0.86201171]
 [ 0.06559617  0.47162993]]
```

Numpy: Array indexing / slicing

In [58]:
```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
a
```

Out[58]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [59]:
```
# 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):
b = a[:2, 1:3]
b
```

Out[59]:
```
array([[2, 3],
       [6, 7]])
```

Numpy: Array indexing / slicing (cont)

In [60]:
```
# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])
```

```
2
```

In [61]:
```
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
```

```
77
```

Numpy: Array indexing / slicing (cont)

In [62]:
```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

In [63]:
```
# Two ways of accessing the data in the middle row of the array. Note the difference in shape
row_r1 = a[1, :]
row_r2 = a[1:2, :]
print(row_r1, row_r1.shape)
```

```
[5 6 7 8] (4,)
```

In [64]:
```
print(row_r2, row_r2.shape)
```

```
[[5 6 7 8]] (1, 4)
```

Numpy: Array indexing / slicing (cont)

In [65]:
```
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
```

```
[ 2  6 10] (3,)
```

In [66]:
```
print(col_r2, col_r2.shape)
```

```
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Numpy: Integer array indexing

In [67]:
```
a = np.array([[1,2], [3, 4], [5, 6]])
```

In [68]:
```
# An example of integer array indexing.
# The returned array will has shape (3,)
print(a[[0, 1, 2], [0, 1, 0]])
```

```
[1 4 5]
```

In [69]:
```
# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[1 4 5]
```

In [70]:
```
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])
```

```
[2 2]
```

In [71]:
```
# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
```

Numpy: Boolean array indexing

In [72]:
```
a = np.array([[1,2], [3, 4], [5, 6]])

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

```
[[False False]
 [ True  True]
 [ True  True]]
```

In [73]:
```
# 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])
```

```
[3 4 5 6]
```

In [74]:
```
# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
[3 4 5 6]
```

Numpy: Element-wise matrix operations

In [75]:
```
x = np.array([[1,2],
              [3,4]], dtype=np.float64)
y = np.array([[5,6],
              [7,8]], dtype=np.float64)

# Elementwise sum
print(x + y)
print(np.add(x, y))
```

```
[[  6.   8.]
 [ 10.  12.]]
[[  6.   8.]
 [ 10.  12.]]
```

In [76]:
```
# Elementwise difference
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

Numpy: Element-wise matrix operations (cont)

In [77]:
```
# Elementwise product
print(x * y)
print(np.multiply(x, y))
```

```
[[  5.  12.]
 [ 21.  32.]]
[[  5.  12.]
 [ 21.  32.]]
```

In [78]:
```
# Elementwise division
print(x / y)
print(np.divide(x, y))
```

```
[[ 0.2         0.33333333]
 [ 0.42857143  0.5       ]]
[[ 0.2         0.33333333]
 [ 0.42857143  0.5       ]]
```

In [79]:
```
# Elementwise square root
print(np.sqrt(x))
```

```
[[ 1.          1.41421356]
 [ 1.73205081  2.        ]]
```

Numpy: Matrix operations

In [80]:
```
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
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

In [81]:
```
# Matrix / vector product
print(x.dot(v))
print(np.dot(x, v))
```

```
[29 67]
[29 67]
```

In [82]:
```
# Matrix / matrix product
print(x.dot(y))
print(np.dot(x, y))
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy: Other useful matrix operations

In [83]:
```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements
```

```
10
```

In [84]:
```
print(np.sum(x, axis=0))  # Compute sum of each column
```

```
[4 6]
```

In [85]:
```
print(np.sum(x, axis=1))  # Compute sum of each row
```

```
[3 7]
```

Numpy: Other useful matrix operations (cont)

In [86]:
```
x = np.array([[1,2], [3,4]])
print(x)
```

```
[[1 2]
 [3 4]]
```

In [87]:
```
print(x.T)
```

```
[[1 3]
 [2 4]]
```

In [88]:
```
# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)
print(v.T)
```

```
[1 2 3]
[1 2 3]
```

Numpy: Other useful matrix operations (cont)

In [89]:
```
x = np.array([[1,2], [3,4]])
print(x.max())
```

```
4
```

In [90]:
```
print(x.min())
```

```
1
```

In [91]:
```
print(x.mean())
```

```
2.5
```

Pandas and high level data structures

Examples from 10 Minutes to pandas

Pandas: Object creation

In [92]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Creating a Series by passing a list of values, letting pandas create a default integer index
s = pd.Series([1,3,5,np.nan,6,8])
s
```

Out[92]:
```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

In [93]:
```
# Creating a DataFrame by passing a numpy array, with a datetime index and labeled columns
dates = pd.date_range('20130101', periods=6)
dates
```

Out[93]:
```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

Pandas: Object creation (cont)

In [94]:
```
df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
df
```

Out[94]:

	A	B	C	D
2013-01-01	1.003134	-0.618860	-0.060117	0.408458
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123
2013-01-03	-1.108226	1.175020	-0.257218	0.310518
2013-01-04	0.347354	0.987915	-0.625398	0.506856
2013-01-05	-0.110601	0.719871	-0.315544	0.507415
2013-01-06	0.929963	-1.600353	-1.803747	-0.063239

In [95]:
```
# Creating a DataFrame by passing a dict of objects that can be converted to series-like.
df2 = pd.DataFrame({ 'A' : 1.,
                     'B' : pd.Timestamp('20130102'),
                     'C' : pd.Series(1,index=list(range(4)),dtype='float32'),
                     'D' : np.array([3] * 4,dtype='int32'),
                     'E' : pd.Categorical(["test","train","test","train"]),
                     'F' : 'foo' })
df2
```

Out[95]:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

Pandas: Viewing Data

In [96]:
```
# See the top & bottom rows of the frame
df.head()
```

Out[96]:

	A	B	C	D
2013-01-01	1.003134	-0.618860	-0.060117	0.408458
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123
2013-01-03	-1.108226	1.175020	-0.257218	0.310518
2013-01-04	0.347354	0.987915	-0.625398	0.506856
2013-01-05	-0.110601	0.719871	-0.315544	0.507415

In [97]:
```
df.tail(3)
```

Out[97]:

	A	B	C	D
2013-01-04	0.347354	0.987915	-0.625398	0.506856
2013-01-05	-0.110601	0.719871	-0.315544	0.507415
2013-01-06	0.929963	-1.600353	-1.803747	-0.063239

Pandas: Viewing Data (cont)

In [98]:
```
# Display the index, columns, and the underlying numpy data
df.index
```

Out[98]:
```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [99]:
```
df.columns
```

Out[99]:
```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

Pandas: Viewing Data (cont)

In [100]:
```
df.values
```

Out[100]:
```
array([[ 1.0031343 , -0.61886008, -0.06011667,  0.40845778],
       [-0.92191951, -0.64119861,  0.32666371, -1.10312278],
       [-1.10822631,  1.17501985, -0.25721843,  0.3105182 ],
       [ 0.34735365,  0.98791479, -0.62539767,  0.50685589],
       [-0.11060075,  0.71987124, -0.31554426,  0.50741525],
       [ 0.92996276, -1.60035276, -1.80374741, -0.06323863]])
```

In [101]:
```
# Describe shows a quick statistic summary of your data
df.describe()
```

Out[101]:

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.023284	0.003732	-0.455893	0.094481
std	0.903199	1.116246	0.731252	0.623691
min	-1.108226	-1.600353	-1.803747	-1.103123
25%	-0.719090	-0.635614	-0.547934	0.030201
50%	0.118376	0.050506	-0.286381	0.359488
75%	0.784310	0.920904	-0.109392	0.482256
max	1.003134	1.175020	0.326664	0.507415

Pandas: Viewing Data (cont)

In [102]:
```
# Transposing your data
df.T
```

Out[102]:

	2013-01-01 00:00:00	2013-01-02 00:00:00	2013-01-03 00:00:00	2013-01-04 00:00:00	2013-01-05 00:00:00	2013-01-06 00:00:00
A	1.003134	-0.921920	-1.108226	0.347354	-0.110601	0.929963
B	-0.618860	-0.641199	1.175020	0.987915	0.719871	-1.600353
C	-0.060117	0.326664	-0.257218	-0.625398	-0.315544	-1.803747
D	0.408458	-1.103123	0.310518	0.506856	0.507415	-0.063239

In [103]:
```
# Sorting by an axis
df.sort_index(axis=1, ascending=False)
```

Out[103]:

	D	C	B	A
2013-01-01	0.408458	-0.060117	-0.618860	1.003134
2013-01-02	-1.103123	0.326664	-0.641199	-0.921920
2013-01-03	0.310518	-0.257218	1.175020	-1.108226
2013-01-04	0.506856	-0.625398	0.987915	0.347354
2013-01-05	0.507415	-0.315544	0.719871	-0.110601
2013-01-06	-0.063239	-1.803747	-1.600353	0.929963

Pandas: Viewing Data (cont)

In [104]:
```
# Sorting by values
df.sort_values(by='B')
```

Out[104]:

	A	B	C	D
2013-01-06	0.929963	-1.600353	-1.803747	-0.063239
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123
2013-01-01	1.003134	-0.618860	-0.060117	0.408458
2013-01-05	-0.110601	0.719871	-0.315544	0.507415
2013-01-04	0.347354	0.987915	-0.625398	0.506856
2013-01-03	-1.108226	1.175020	-0.257218	0.310518

Pandas: Selection

In [105]:
```
# Selecting a single column, which yields a Series, equivalent to df.A
df['A']
```

Out[105]:
```
2013-01-01    1.003134
2013-01-02   -0.921920
2013-01-03   -1.108226
2013-01-04    0.347354
2013-01-05   -0.110601
2013-01-06    0.929963
Freq: D, Name: A, dtype: float64
```

In [106]:
```
# Selecting via [], which slices the rows.
df[0:3]
```

Out[106]:

	A	B	C	D
2013-01-01	1.003134	-0.618860	-0.060117	0.408458
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123
2013-01-03	-1.108226	1.175020	-0.257218	0.310518

In [107]:
```
df['20130102':'20130104']
```

Out[107]:

	A	B	C	D
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123
2013-01-03	-1.108226	1.175020	-0.257218	0.310518
2013-01-04	0.347354	0.987915	-0.625398	0.506856

Pandas: Selection (cont)

In [108]:
```
# For getting a cross section using a label
df.loc[dates[0]]
```

Out[108]:
```
A    1.003134
B   -0.618860
C   -0.060117
D    0.408458
Name: 2013-01-01 00:00:00, dtype: float64
```

In [109]:
```
# Selecting on a multi-axis by label
df.loc[:,['A','B']]
```

Out[109]:

	A	B
2013-01-01	1.003134	-0.618860
2013-01-02	-0.921920	-0.641199
2013-01-03	-1.108226	1.175020
2013-01-04	0.347354	0.987915
2013-01-05	-0.110601	0.719871
2013-01-06	0.929963	-1.600353

Pandas: Selection (cont)

In [110]:
```
# Showing label slicing, both endpoints are included
df.loc['20130102':'20130104',['A','B']]
```

Out[110]:

	A	B
2013-01-02	-0.921920	-0.641199
2013-01-03	-1.108226	1.175020
2013-01-04	0.347354	0.987915

In [111]:
```
# Reduction in the dimensions of the returned object
df.loc['20130102',['A','B']]
```

Out[111]:
```
A   -0.921920
B   -0.641199
Name: 2013-01-02 00:00:00, dtype: float64
```

In [112]:
```
# For getting a scalar value
df.loc[dates[0],'A']
```

Out[112]:
```
1.0031343031279012
```

Pandas: Selection (cont)

In [113]:
```
# For getting fast access to a scalar (equiv to the prior method)
df.at[dates[0],'A']
```

Out[113]:
```
1.0031343031279012
```

In [114]:
```
# Select via the position of the passed integers
df.iloc[3]
```

Out[114]:
```
A    0.347354
B    0.987915
C   -0.625398
D    0.506856
Name: 2013-01-04 00:00:00, dtype: float64
```

In [115]:
```
# By integer slices, acting similar to numpy/python
df.iloc[3:5,0:2]
```

Out[115]:

	A	B
2013-01-04	0.347354	0.987915
2013-01-05	-0.110601	0.719871

Pandas: Selection (cont)

In [116]:
```
# By lists of integer position locations, similar to the numpy/python style
df.iloc[[1,2,4],[0,2]]
```

Out[116]:

	A	C
2013-01-02	-0.921920	0.326664
2013-01-03	-1.108226	-0.257218
2013-01-05	-0.110601	-0.315544

In [117]:
```
# For slicing rows explicitly
df.iloc[1:3,:]
```

Out[117]:

	A	B	C	D
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123
2013-01-03	-1.108226	1.175020	-0.257218	0.310518

Pandas: Selection (cont)

In [118]:
```
# For slicing columns explicitly
df.iloc[:,1:3]
```

Out[118]:

	B	C
2013-01-01	-0.618860	-0.060117
2013-01-02	-0.641199	0.326664
2013-01-03	1.175020	-0.257218
2013-01-04	0.987915	-0.625398
2013-01-05	0.719871	-0.315544
2013-01-06	-1.600353	-1.803747

In [119]:
```
# For getting a value explicitly
df.iloc[1,1]
```

Out[119]:
```
-0.64119860868718015
```

In [120]:
```
# For getting fast access to a scalar (equiv to the prior method)
df.iat[1,1]
```

Out[120]:
```
-0.64119860868718015
```

Pandas: Boolean Indexing

In [121]:
```
# Using a single column's values to select data.
df[df.A > 0]
```

Out[121]:

	A	B	C	D
2013-01-01	1.003134	-0.618860	-0.060117	0.408458
2013-01-04	0.347354	0.987915	-0.625398	0.506856
2013-01-06	0.929963	-1.600353	-1.803747	-0.063239

In [122]:
```
# Selecting values from a DataFrame where a boolean condition is met.
df[df > 0]
```

Out[122]:

	A	B	C	D
2013-01-01	1.003134	NaN	NaN	0.408458
2013-01-02	NaN	NaN	0.326664	NaN
2013-01-03	NaN	1.175020	NaN	0.310518
2013-01-04	0.347354	0.987915	NaN	0.506856
2013-01-05	NaN	0.719871	NaN	0.507415
2013-01-06	0.929963	NaN	NaN	NaN

Pandas: Boolean Indexing (cont)

In [123]:
```
# Using the isin() method for filtering:
df2 = df.copy()
df2['E'] = ['one', 'one','two','three','four','three']
df2
```

Out[123]:

	A	B	C	D	E
2013-01-01	1.003134	-0.618860	-0.060117	0.408458	one
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123	one
2013-01-03	-1.108226	1.175020	-0.257218	0.310518	two
2013-01-04	0.347354	0.987915	-0.625398	0.506856	three
2013-01-05	-0.110601	0.719871	-0.315544	0.507415	four
2013-01-06	0.929963	-1.600353	-1.803747	-0.063239	three

In [124]:
```
df2[df2['E'].isin(['two','four'])]
```

Out[124]:

	A	B	C	D	E
2013-01-03	-1.108226	1.175020	-0.257218	0.310518	two
2013-01-05	-0.110601	0.719871	-0.315544	0.507415	four

Pandas: Missing data

In [125]:
```
# Here we modify our dataset to explore handling missing
s1 = pd.Series([1,2,3,4,5,6], index=pd.date_range('20130102', periods=6))
df['F'] = s1
df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ['E'])
df1.loc[dates[0]:dates[1],'E'] = 1
df1
```

Out[125]:

	A	B	C	D	F	E
2013-01-01	1.003134	-0.618860	-0.060117	0.408458	NaN	1.0
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123	1.0	1.0
2013-01-03	-1.108226	1.175020	-0.257218	0.310518	2.0	NaN
2013-01-04	0.347354	0.987915	-0.625398	0.506856	3.0	NaN

In [126]:
```
# To drop any rows that have missing data.
df1.dropna(how='any')
```

Out[126]:

	A	B	C	D	F	E
2013-01-02	-0.92192	-0.641199	0.326664	-1.103123	1.0	1.0

Pandas: Missing data (cont)

In [127]:
```
# Filling missing data
df1.fillna(value=5)
```

Out[127]:

	A	B	C	D	F	E
2013-01-01	1.003134	-0.618860	-0.060117	0.408458	5.0	1.0
2013-01-02	-0.921920	-0.641199	0.326664	-1.103123	1.0	1.0
2013-01-03	-1.108226	1.175020	-0.257218	0.310518	2.0	5.0
2013-01-04	0.347354	0.987915	-0.625398	0.506856	3.0	5.0

In [128]:
```
# To get the boolean mask where values are nan
df1.isnull()
```

Out[128]:

	A	B	C	D	F	E
2013-01-01	False	False	False	False	True	False
2013-01-02	False	False	False	False	False	False
2013-01-03	False	False	False	False	False	True
2013-01-04	False	False	False	False	False	True

Pandas: Operations

In [129]:
```
# Performing a descriptive statistic
df.mean()
```

Out[129]:
```
A    0.023284
B    0.003732
C   -0.455893
D    0.094481
F    3.000000
dtype: float64
```

In [130]:
```
# Same operation on the other axis
df.mean(1)
```

Out[130]:
```
2013-01-01    0.183154
2013-01-02   -0.267915
2013-01-03    0.424019
2013-01-04    0.843345
2013-01-05    0.960228
2013-01-06    0.492525
Freq: D, dtype: float64
```

In [131]:
```
# Apply functions to data
df.apply(np.cumsum)
```

Out[131]:

	A	B	C	D	F
2013-01-01	1.003134	-0.618860	-0.060117	0.408458	NaN
2013-01-02	0.081215	-1.260059	0.266547	-0.694665	1.0
2013-01-03	-1.027012	-0.085039	0.009329	-0.384147	3.0
2013-01-04	-0.679658	0.902876	-0.616069	0.122709	6.0
2013-01-05	-0.790259	1.622747	-0.931613	0.630124	10.0
2013-01-06	0.139704	0.022394	-2.735361	0.566886	15.0

In [132]:
```
df.apply(lambda x: x.max() - x.min())
```

Out[132]:
```
A    2.111361
B    2.775373
C    2.130411
D    1.610538
F    4.000000
dtype: float64
```

Pandas: Histogramming

In [133]:
```
s = pd.Series(np.random.randint(0, 7, size=10))
s
```

Out[133]:
```
0    0
1    3
2    6
3    5
4    4
5    4
6    5
7    6
8    6
9    3
dtype: int64
```

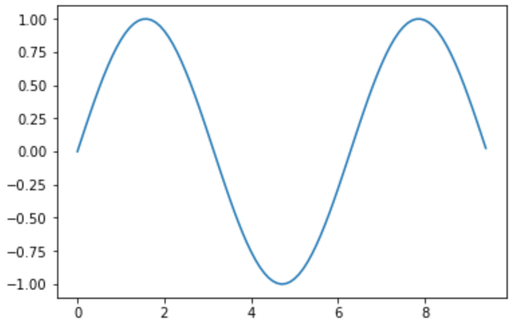
In [134]:
```
s.value_counts()
```

Out[134]:
```
6    3
5    2
4    2
3    2
0    1
dtype: int64
```

Plotting and Matplotlib

Matplotlib: Example plot

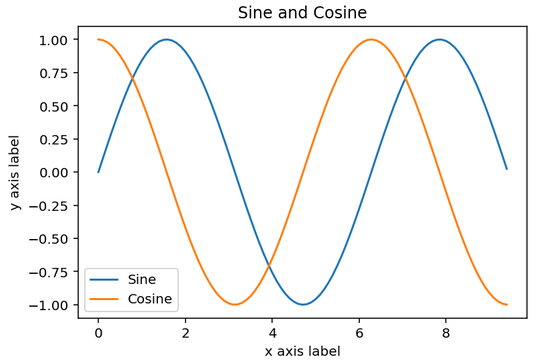
In [135]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```



Matplotlib: Add multiple lines to one plot

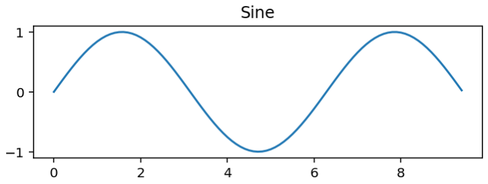
In [136]:
```
%config InlineBackend.figure_format = 'retina'
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
```

In [137]:
```
# Plot the points using matplotlib
plt.plot(x, y_sin, label='Sine')
plt.plot(x, y_cos, label='Cosine')
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend()
plt.show()
```



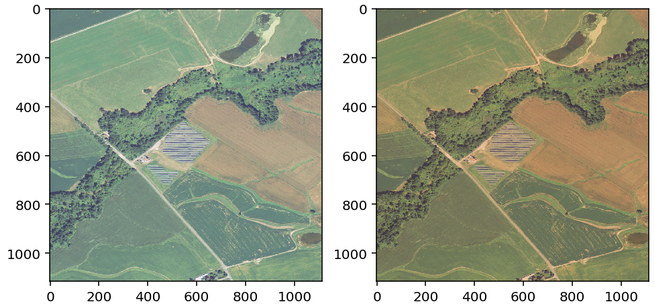
Matplotlib: Subplots

In [138]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')
plt.show()

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



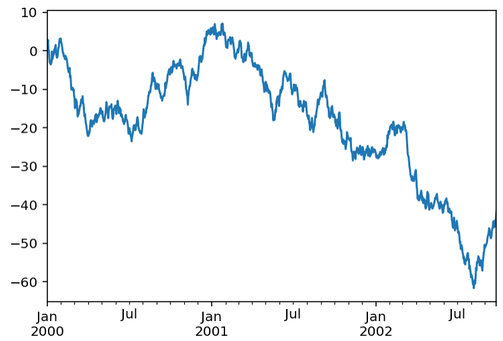
Matplotlib: Images

In [139]:
```
from scipy.misc import imread, imresize

img = imread('datasets/images/plant1.tif')
img_tinted = img * [1, 0.85, 0.7]

# Show the original image
plt.figure(figsize=(8,4))
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results
# if presented with data that is not uint8. To work around this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```



Pandas: plotting

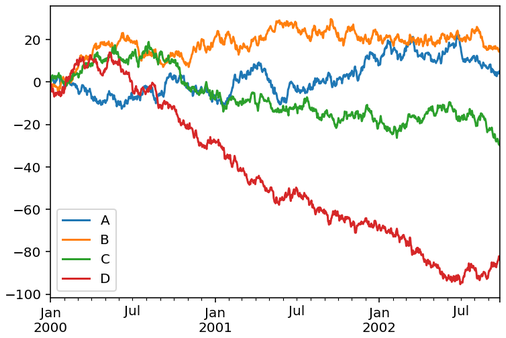
In [140]:
```
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
ts = ts.cumsum()
ts.plot()
plt.show()
```



Pandas: plotting (cont)

In [141]:
```
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
                  columns=['A', 'B', 'C', 'D'])
df = df.cumsum()
plt.figure();
df.plot();
plt.legend(loc='best')
plt.show()
```

```
<matplotlib.figure.Figure at 0x11322f4a8>
```



Pandas: Loading CSV data

In [142]:
```
df = pd.read_csv('datasets/housing/housing.csv')
df
```

Out[142]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY
5	-122.25	37.85	52.0	919.0	213.0	413.0	193.0	4.0368	269700.0	NEAR BAY
6	-122.25	37.84	52.0	2535.0	489.0	1094.0	514.0	3.6591	299200.0	NEAR BAY
7	-122.25	37.84	52.0	3104.0	687.0	1157.0	647.0	3.1200	241400.0	NEAR BAY
8	-122.26	37.84	42.0	2555.0	665.0	1206.0	595.0	2.0804	226700.0	NEAR BAY
9	-122.25	37.84	52.0	3549.0	707.0	1551.0	714.0	3.6912	261100.0	NEAR BAY
10	-122.26	37.85	52.0	2202.0	434.0	910.0	402.0	3.2031	281500.0	NEAR BAY
11	-122.26	37.85	52.0	3503.0	752.0	1504.0	734.0	3.2705	241800.0	NEAR BAY
12	-122.26	37.85	52.0	2491.0	474.0	1098.0	468.0	3.0750	213500.0	NEAR BAY
13	-122.26	37.84	52.0	696.0	191.0	345.0	174.0	2.6736	191300.0	NEAR BAY
14	-122.26	37.85	52.0	2643.0	626.0	1212.0	620.0	1.9167	159200.0	NEAR BAY
15	-122.26	37.85	50.0	1120.0	283.0	697.0	264.0	2.1250	140000.0	NEAR BAY
16	-122.27	37.85	52.0	1966.0	347.0	793.0	331.0	2.7750	152500.0	NEAR BAY
17	-122.27	37.85	52.0	1228.0	293.0	648.0	303.0	2.1202	155500.0	NEAR BAY
18	-122.26	37.84	50.0	2239.0	455.0	990.0	419.0	1.9911	158700.0	NEAR BAY
19	-122.27	37.84	52.0	1503.0	298.0	690.0	275.0	2.6033	162900.0	NEAR BAY
20	-122.27	37.85	40.0	751.0	184.0	409.0	166.0	1.3578	147500.0	NEAR BAY
21	-122.27	37.85	42.0	1639.0	367.0	929.0	366.0	1.7135	159800.0	NEAR BAY
22	-122.27	37.84	52.0	2436.0	541.0	1015.0	478.0	1.7250	113900.0	NEAR BAY
23	-122.27	37.84	52.0	1688.0	337.0	853.0	325.0	2.1806	99700.0	NEAR BAY
24	-122.27	37.84	52.0	2224.0	437.0	1006.0	422.0	2.6000	132600.0	NEAR BAY
25	-122.28	37.85	41.0	535.0	123.0	317.0	119.0	2.4038	107500.0	NEAR BAY
26	-122.28	37.85	49.0	1130.0	244.0	607.0	239.0	2.4597	93800.0	NEAR BAY
27	-122.28	37.85	52.0	1898.0	421.0	1102.0	397.0	1.8080	105500.0	NEAR BAY
28	-122.28	37.84	50.0	2082.0	492.0	1131.0	473.0	1.6424	108900.0	NEAR BAY
29	-122.28	37.84	52.0	729.0	160.0	395.0	155.0	1.6875	132000.0	NEAR BAY
...	...	...	...	...	...	...	...	...	...	...
20610	-121.56	39.10	28.0	2130.0	484.0	1195.0	439.0	1.3631	45500.0	INLAND
20611	-121.55	39.10	27.0	1783.0	441.0	1163.0	409.0	1.2857	47000.0	INLAND
20612	-121.56	39.08	26.0	1377.0	289.0	761.0	267.0	1.4934	48300.0	INLAND
20613	-121.55	39.09	31.0	1728.0	365.0	1167.0	384.0	1.4958	53400.0	INLAND
20614	-121.54	39.08	26.0	2276.0	460.0	1455.0	474.0	2.4695	58000.0	INLAND
20615	-121.54	39.08	23.0	1076.0	216.0	724.0	197.0	2.3598	57500.0	INLAND
20616	-121.53	39.08	15.0	1810.0	441.0	1157.0	375.0	2.0469	55100.0	INLAND
20617	-121.53	39.06	20.0	561.0	109.0	308.0	114.0	3.3021	70800.0	INLAND
20618	-121.55	39.06	25.0	1332.0	247.0	726.0	226.0	2.2500	63400.0	INLAND