The Numpy array object

Section contents

What are Numpy and Numpy arrays?

Python objects

- high-level number objects: integers, floating point
- containers: lists (costless insertion and append), dictionaries (fast lookup)

Numpy provides

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)
- Also known as *array oriented computing*

In [1]:
```
import numpy as np
a = np.array([0, 1, 2, 3])
a
```

Out[1]:
```
array([0, 1, 2, 3])
```

For example, An array containing:

- values of an experiment/simulation at discrete time steps
- signal recorded by a measurement device, e.g. sound wave
- pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

Why it is useful: Memory-efficient container that provides fast numerical operations.

In [2]:
```
L = range(1000)
```

In [3]:
```
%timeit [i**2 for i in L]
```

```
10000 loops, best of 3: 71.6 µs per loop
```

In [4]:
```
a = np.arange(1000)
```

In [5]:
```
%timeit a**2
```

```
100000 loops, best of 3: 1.78 µs per loop
```

Reference documentation

- On the web: http://docs.scipy.org (http://docs.scipy.org)/
- Interactive help:

In [6]:
```
np.array?
```

- Looking for something:

In [7]:
```
np.lookfor('create array')
```

```
Search results for 'create array'
---------------------------------
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
numpy.diagflat
    Create a two-dimensional array with the flattened input as a diagonal.
numpy.fromiter
    Create a new 1-dimensional array from an iterable object.
numpy.partition
    Return a partitioned copy of an array.
numpy.ma.diagflat
    Create a two-dimensional array with the flattened input as a diagonal.
numpy.ctypeslib.as_array
    Create a numpy array from a ctypes array or a ctypes POINTER.
numpy.ma.make_mask
    Create a boolean mask from an array.
numpy.ctypeslib.as_ctypes
    Create and return a ctypes object from a numpy array.  Actually
numpy.ma.mrecords.fromarrays
    Creates a mrecarray from a (flat) list of masked arrays.
numpy.lib.format.open_memmap

    Open a .npy file as a memory-mapped array.
numpy.ma.MaskedArray.__new__
    Create a new masked array from scratch.
numpy.lib.arrayterator.Arrayterator
    Buffered iterator for big arrays.
numpy.ma.mrecords.fromtextfile
    Creates a mrecarray from data stored in the file `filename`.
numpy.oldnumeric.ma.fromfunction
    apply f to s to create array as in umath.
numpy.oldnumeric.ma.masked_object
    Create array masked where exactly data equal to value
numpy.oldnumeric.ma.masked_values
    Create a masked array; mask is nomask if possible.
numpy.asarray
    Convert the input to an array.
numpy.ndarray
    ndarray(shape, dtype=float, buffer=None, offset=0,
numpy.recarray
    Construct an ndarray that allows field access using attributes.
numpy.chararray
    chararray(shape, itemsize=1, unicode=False, buffer=None, offset=0,
numpy.pad
    Pads an array.
numpy.sum
    Sum of array elements over a given axis.
numpy.asanyarray
    Convert the input to an ndarray, but pass ndarray subclasses through.
```

numpy.copy
  Return an array copy of the given object.
numpy.diag
  Extract a diagonal or construct a diagonal array.
numpy.load
  Load an array(s) or pickled objects from .npy, .npz, or pickled files.
numpy.sort
  Return a sorted copy of an array.
numpy.array_equiv
  Returns True if input arrays are shape consistent and all elements equal.
numpy.dtype
  Create a data type object.
numpy.choose
  Construct an array from an index array and a set of arrays to choose from.
numpy.nditer
  Efficient multi-dimensional iterator object to iterate over arrays.
numpy.swapaxes
  Interchange two axes of an array.
numpy.full_like
  Return a full array with the same shape and type as a given array.
numpy.ones_like
  Return an array of ones with the same shape and type as a given array.
numpy.empty_like
  Return a new array with the same shape and type as a given array.
numpy.zeros_like
  Return an array of zeros with the same shape and type as a given array.
numpy.asarray_chkfinite
  Convert the input to an array, checking for NaNs or Infs.
numpy.diag_indices
  Return the indices to access the main diagonal of an array.
numpy.ma.choose
  Use an index array to construct a new array from a set of choices.
numpy.chararray.tolist
  a.tolist()
numpy.matlib.rand
  Return a matrix of random values with given shape.
numpy.savez_compressed
  Save several arrays into a single file in compressed ``.npz`` format.
numpy.ma.empty_like
  Return a new array with the same shape and type as a given array.
numpy.ma.make_mask_none
  Return a boolean mask of the given shape, filled with False.
numpy.ma.mrecords.fromrecords
  Creates a MaskedRecords from a list of records.
numpy.around
  Evenly round to the given number of decimals.
numpy.source
  Print or write to a file the source code for a Numpy object.
numpy.diagonal
  Return specified diagonals.
numpy.histogram2d
  Compute the bi-dimensional histogram of two data samples.
numpy.fft.ifft
  Compute the one-dimensional inverse discrete Fourier Transform.
numpy.fft.ifftn
  Compute the N-dimensional inverse discrete Fourier Transform.
numpy.busdaycalendar
  A business day calendar object that efficiently stores information

In [10]:
```
np.con*?
```

Import conventions

The general convention to import numpy is:

In [11]:
```
import numpy as np
```

Using this style of import is recommended.

Creating arrays

- **1-D**:

In [12]:
```
a = np.array([0, 1, 2, 3])
a
```

Out[12]:
```
array([0, 1, 2, 3])
```

In [13]:
```
a.ndim
```

Out[13]:
```
1
```

In [14]:
```
a.shape
```

Out[14]:
```
(4,)
```

In [15]:
```
len(a)
```

Out[15]:
```
4
```

- **2-D, 3-D, ...**:

In [16]:
```
b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
b
```

Out[16]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [17]:
```
b.ndim
```

Out[17]:
```
2
```

In [18]:
```
b.shape
```

Out[18]:
```
(2, 3)
```

In [19]:
```
len(b)     # returns the size of the first dimension
```

Out[19]:
```
2
```

In [20]:
```
c = np.array([[[1], [2]], [[3], [4]]])
c
```

Out[20]:
```
array([[[1],
        [2]],

       [[3],
        [4]]])
```

In [21]:
```
c.shape
```

Out[21]:
```
(2, 2, 1)
```

Exercise: Simple arrays

- Create simple one and two dimensional arrays. First, redo the examples from above. And then create your own.
- Use the functions len, shape and ndim on some of those arrays and observe their output.

Functions for creating arrays

In practice, we rarely enter items one by one...

- Evenly spaced:

In [22]:
```
a = np.arange(10) # 0 .. n-1  (!)
a
```

Out[22]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [23]:
```
b = np.arange(1, 9, 2) # start, end (exclusive), step
b
```

Out[23]:
```
array([1, 3, 5, 7])
```

- or by number of points:

In [24]:
```
c = np.linspace(0, 1, 6)   # start, end, num-points
c
```

Out[24]:
```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

In [25]:
```
d = np.linspace(0, 1, 5, endpoint=False)
d
```

Out[25]:
```
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

- Common arrays:

In [26]:
```
a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
a
```

Out[26]:
```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

In [27]:
```
b = np.zeros((2, 2))
b
```

Out[27]:
```
array([[ 0.,  0.],
       [ 0.,  0.]])
```

In [28]:
```
c = np.eye(3)
c
```

Out[28]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

In [29]:
```
d = np.diag(np.array([1, 2, 3, 4]))
d
```

Out[29]:
```
array([[1, 0, 0, 0],
```

Out[29]: array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])

- np.random random numbers (Mersenne Twister PRNG):

In [30]:
```
a = np.random.rand(4)       # uniform in [0, 1]
a
```

Out[30]: array([0.90103456, 0.04550118, 0.26387871, 0.42707105])

In [31]:
```
b = np.random.randn(4)      # Gaussian
b
```

Out[31]: array([-0.24637953, 0.5029874 , 1.18003342, 0.06942228])

In [32]:
```
np.random.seed(1234)        # Setting the random seed
```

Exercise: Creating arrays using functions

- Experiment with arange, linspace, ones, zeros, eye and diag.
- Create different kinds of arrays with random numbers.
- Try setting the seed before creating an array with random values.
- Look at the function np.empty. What does it do? When might this be useful?

Basic data types

You may have noticed that, in some instances, array elements are displayed with a trailing dot (e.g. 2. vs 2). This is due to a difference in the data-type used:

In [33]:
```
a = np.array([1, 2, 3])
a.dtype
```

Out[33]: dtype('int64')

In [34]:
```
b = np.array([1., 2., 3.])
b.dtype
```

Out[34]: dtype('float64')

Tip

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

You can explicitly specify which data-type you want:

In [35]:
```
c = np.array([1, 2, 3], dtype=float)
c.dtype
```

Out[35]: dtype('float64')

The **default** data type is floating point:

In [36]:
```
a = np.ones((3, 3))
a.dtype
```

Out[36]: dtype('float64')

There are also other types:

Complex

In [37]:
```
d = np.array([1+2j, 3+4j, 5+6*1j])
d.dtype
```

Out[37]: dtype('complex128')

Bool

In [38]:
```
e = np.array([True, False, False, True])
e.dtype
```

Out[38]: dtype('bool')

Strings

In [39]:
```
f = np.array(['Bonjour', 'Hello', 'Hallo',])
f.dtype     # <--- strings containing max. 7 letters
```

Out[39]: dtype('S7')

Much more

- int32
- int64
- unit32
- unit64

Basic visualization

Tip

Now that we have our first data arrays, we are going to visualize them.

Start by launching IPython in *pylab* mode.

```
$ ipython --pylab
```

Or the notebook:

```
$ ipython notebook --pylab=inline
```

Alternatively, if IPython has already been started:

In [40]:
```
%pylab
```

```
Using matplotlib backend: TkAgg
Populating the interactive namespace from numpy and matplotlib

WARNING: pylab import has clobbered these variables: ['e', 'f']
`%pylab --no-import-all` prevents importing * from pylab and numpy
```

Or, from the notebook:

In []:
```
%pylab inline
```

The inline is important for the notebook, so that plots are displayed in the notebook and not in a new window.

Matplotlib is a 2D plotting package. We can import its functions as below:

In [41]:
```
import matplotlib.pyplot as plt  # the tidy way
```

And then use (note that you have to use show explicitly):

In [42]:
```
plt.plot(x, y)       # line plot
```

```
plt.show()           # <-- shows the plot (not needed with pylab)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-42-cca45a0ba107> in <module>()
----> 1 plt.plot(x, y)       # line plot
      2 plt.show()           # <-- shows the plot (not needed with pylab)

NameError: name 'x' is not defined
```

Or, if you are using *pylab*:

In []:
```
plot(x, y)       # line plot
```

Using import matplotlib.pyplot as plt is recommended for use in scripts. Whereas pylab is recommended for interactive exploratory work.

- **1D plotting**:

In []:
```
x = np.linspace(0, 3, 20)
y = np.linspace(0, 9, 20)
plt.plot(x, y)       # line plot
```

In []:
```
plt.plot(x, y, 'o')  # dot plot
```

- **2D arrays** (such as images):

In []:
```
image = np.random.rand(30, 30)
plt.imshow(image, cmap=plt.cm.hot)
plt.colorbar()
```

More in the Matplotlib tutorial this afternoon

Exercise: Simple visualizations

- Plot some simple arrays.
- Try to use both the IPython shell and the notebook, if possible.
- Try using the gray colormap.

Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

In [43]:
```
a = np.arange(10)
a
```

Out[43]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [44]:
```
a[0], a[2], a[-1]
```

Out[44]:
```
(0, 2, 9)
```

Warning

Indices begin at 0, like other Python sequences (and C/C++). In contrast, in Fortran or Matlab, indices begin at 1.

The usual python idiom for reversing a sequence is supported:

In [45]:
```
a[::-1]
```

Out[45]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

For multidimensional arrays, indexes are tuples of integers:

In [46]:
```
a = np.diag(np.arange(3))
a
```

Out[46]:
```
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
```

In [47]:
```
a[1, 1]
```

Out[47]:
```
1
```

In [48]:
```
a[2, 1] = 10 # third line, second column
a
```

Out[48]:
```
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
```

In [49]:
```
a[1]
```

Out[49]:
```
array([0, 1, 0])
```

Note that:

- In 2D, the first dimension corresponds to rows, the second to columns.
- Let us repeat together: the first dimension corresponds to **rows**, the second to **columns**.
- for multidimensional a, a[0] is interpreted by taking all elements in the unspecified dimensions.

Slicing Arrays, like other Python sequences can also be sliced:

In [50]:
```
a = np.arange(10)
a
```

Out[50]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [51]:
```
a[2:9:3] # [start:end:step]
```

Out[51]:
```
array([2, 5, 8])
```

Note that the last index is not included! :

In [52]:
```
a[:4]
```

Out[52]:
```
array([0, 1, 2, 3])
```

All three slice components are not required: by default, `start` is 0, `end` is the last and `step` is 1:

In [53]:
```
a[1:3]
```

Out[53]:
```
array([1, 2])
```

In [54]:
```
a[::2]
```

Out[54]:
```
array([0, 2, 4, 6, 8])
```

In [55]:
```
a[3:]
```

Out[55]:
```
array([3, 4, 5, 6, 7, 8, 9])
```

A small illustrated summary of Numpy indexing and slicing...

In [56]:
```
from IPython.display import Image
Image(filename='images/numpy_indexing.png')
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-56-ef00be976d20> in <module>()
      1 from IPython.display import Image
----> 2 Image(filename='images/numpy_indexing.png')

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in __init__(self, data, url, filename, format, embed, width, height, retina)
    599         self.height = height
    600         self.retina = retina
--> 601         super(Image, self).__init__(data=data, url=url, filename=filename)
    602 
    603         if retina:

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in __init__(self, data, url, filename)
    303         self.filename = None if filename is None else unicode(filename)
    304 
--> 305         self.reload()
    306 
    307     def reload(self):

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in reload(self)
    621         """Reload the raw data from file or URL."""
    622         if self.embed:
--> 623             super(Image,self).reload()
    624             if self.retina:
    625                 self._retina_shape()

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in reload(self)
    308         """Reload the raw data from file or URL."""
    309         if self.filename is not None:
--> 310             with open(self.filename, self._read_flags) as f:
    311                 self.data = f.read()
    312         elif self.url is not None:

IOError: [Errno 2] No such file or directory: u'images/numpy_indexing.png'
```

You can also combine assignment and slicing:

In [57]:
```
a = np.arange(10)
a[5:] = 10
a
```

Out[57]:
```
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
```

In [58]:
```
b = np.arange(5)
a[5:] = b[::-1]
a
```

Out[58]:
```
array([0, 1, 2, 3, 4, 4, 3, 2, 1, 0])
```

Exercise: Indexing and slicing

- Try the different flavours of slicing, using start, end and step.
- Verify that the slices in the diagram above are indeed correct. You may use the following expression to create the array:

In [59]:
```
np.arange(6) + np.arange(0, 51, 10)[:, np.newaxis]
```

Out[59]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

- Try assigning a smaller 2D array to a larger 2D array, like in the 1D example above.
- Use a different step, e.g. -2, in the reversal idiom above. What effect does this have?

Exercise: Array creation

Create the following arrays (with correct data types):

[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 2], [1, 6, 1, 1]] [[0., 0., 0., 0., 0.], [2., 0., 0., 0., 0.], [0., 3., 0., 0., 0.], [0., 0., 4., 0., 0.], [0., 0., 0., 5., 0.], [0., 0., 0., 0., 6.]]

Par on course: 3 statements for each

Hint: Individual array elements can be accessed similarly to a list, e.g. a[1] or a[1, 2].

Hint: Examine the docstring for diag.

Exercise: Tiling for array creation

Skim through the documentation for np.tile, and use this function to construct the array:

[[4, 3, 4, 3, 4, 3], [2, 1, 2, 1, 2, 1], [4, 3, 4, 3, 4, 3], [2, 1, 2, 1, 2, 1]]

Copies and views

A slicing operation creates a **view** on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use np.may_share_memory() to check if two arrays share the same memory block. Note however, that this uses heuristics and may give you false positives.

When modifying the view, the original array is modified as well:

In [60]:
```
a = np.arange(10)
a
```

Out[60]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [61]:
```
b = a[::2]
b
```

Out[61]: array([0, 2, 4, 6, 8])

In [62]:
```
np.may_share_memory(a, b)
```

Out[62]: True

In [63]:
```
b[0] = 12
b
```

Out[63]: array([12, 2, 4, 6, 8])

In [64]:
```
a   # (!)
```

Out[64]: array([12, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [65]:
```
a = np.arange(10)
c = a[::2].copy()  # force a copy
c[0] = 12
a
```

Out[65]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [66]:
```
np.may_share_memory(a, c)
```

Out[66]: False

This behavior can be surprising at first sight... but it allows to save both memory and time.

Worked example: Prime number sieve

In [67]:
```
from IPython.display import Image
Image(filename='images/prime-sieve.png')
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-67-08ed76b99731> in <module>()
      1 from IPython.display import Image
----> 2 Image(filename='images/prime-sieve.png')

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in __init__(self, data, url, filename, format, embed, width, height, retina)
    599         self.height = height
    600         self.retina = retina
--> 601         super(Image, self).__init__(data=data, url=url, filename=filename)
    602 
    603         if retina:

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in __init__(self, data, url, filename)
    303         self.filename = None if filename is None else unicode(filename)
    304 
--> 305         self.reload()
    306 
    307     def reload(self):

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in reload(self)
    621         """Reload the raw data from file or URL."""
    622         if self.embed:
--> 623             super(Image,self).reload()
    624             if self.retina:
    625                 self._retina_shape()

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in reload(self)
    308         """Reload the raw data from file or URL."""
    309         if self.filename is not None:
--> 310             with open(self.filename, self._read_flags) as f:
    311                 self.data = f.read()
    312         elif self.url is not None:

IOError: [Errno 2] No such file or directory: u'images/prime-sieve.png'
```

Compute prime numbers in 0--99, with a sieve

- Construct a shape (100,) boolean array is_prime, filled with True in the beginning:

In []:
```
is_prime = np.ones((100,), dtype=bool)
```

- Cross out 0 and 1 which are not primes:

In []:
```
is_prime[:2] = 0
```

- For each integer j starting from 2, cross out its higher multiples:

In []:
```
N_max = int(np.sqrt(len(is_prime)))
for j in range(2, N_max):
    is_prime[2*j::j] = False
```

- Skim through help(np.nonzero), and print the prime numbers
- Follow-up:
  - Move the above code into a script file named prime_sieve.py
  - Run it to check it works
  - Use the optimization suggested in the sieve of Eratosthenes (http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes):
  - Skip j which are already known to not be primes
  - The first number to cross out is \(j^2\)

Fancy indexing

Tip

Numpy arrays can be indexed with slices, but also with boolean or integer arrays (**masks**). This method is called *fancy indexing*. It creates **copies not views**.

Using boolean masks

In [68]:
```
np.random.seed(3)
a = np.random.random_integers(0, 20, 15)
a
```

Out[68]:
```
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
```

In [69]:
```
(a % 3 == 0)
```

Out[69]:
```
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
```

In [70]:
```
mask = (a % 3 == 0)
extract_from_a = a[mask] # or,  a[a%3==0]
extract_from_a           # extract a sub-array with the mask
```

Out[70]:
```
array([ 3,  0,  9,  6,  0, 12])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

In [71]:
```
a[a % 3 == 0] = -1
a
```

Out[71]:
```
array([10, -1,  8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1,  7, 14])
```

Indexing with an array of integers

In [72]:
```
a = np.arange(0, 100, 10)
a
```

Out[72]:
```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Indexing can be done with an array of integers, where the same index is repeated several time:

In [73]:
```
a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
```

Out[73]:
```
array([20, 30, 20, 40, 20])
```

New values can be assigned with this kind of indexing:

In [74]:
```
a[[9, 7]] = -100
a
```

Out[74]: array([0,   10,   20,   30,   40,   50,   60, -100,   80, -100])

Tip

When a new array is created by indexing with an array of integers, the new array has the same shape than the array of integers:

In [75]:
```
a = np.arange(10)
idx = np.array([[3, 4], [9, 7]])
idx.shape
```

Out[75]: (2, 2)

In [76]:
```
a[idx]
```

Out[76]:
```
array([[3, 4],
       [9, 7]])
```

The image below illustrates various fancy indexing applications

In [77]:
```
from IPython.display import Image
Image(filename='images/numpy_fancy_indexing.png')
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-77-beb616f120d9> in <module>()
      1 from IPython.display import Image
----> 2 Image(filename='images/numpy_fancy_indexing.png')

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in __init__(self, data, url, filename, format, embed, width, height, retina)
    599         self.height = height
    600         self.retina = retina
--> 601         super(Image, self).__init__(data=data, url=url, filename=filename)
    602 
    603         if retina:

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in __init__(self, data, url, filename)
    303         self.filename = None if filename is None else unicode(filename)
    304 
--> 305         self.reload()
    306 
    307     def reload(self):

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in reload(self)
    621         """Reload the raw data from file or URL."""
    622         if self.embed:
--> 623             super(Image,self).reload()
    624             if self.retina:
    625                 self._retina_shape()

/usr/lib/python2.7/dist-packages/IPython/core/display.pyc in reload(self)
    308         """Reload the raw data from file or URL."""
    309         if self.filename is not None:
--> 310             with open(self.filename, self._read_flags) as f:
    311                 self.data = f.read()
    312         elif self.url is not None:

IOError: [Errno 2] No such file or directory: u'images/numpy_fancy_indexing.png'
```

Exercise: Fancy indexing

- Again, verify the fancy indexing shown in the diagram above.

- Use fancy indexing on the left and array creation on the right to assign values from a smaller array to a larger array.