Lecture Notes 2: Numpy, Timing, Plotting

Numpy

Basics

```
In [1]: # Import the module such that we can use the built-in functionality
        import numpy
```

Numpy arrays

```
In [2]: X = numpy.array([1, 2, 3, 4])
        Y = numpy.array([5, 6, 7, 8])
```

Operations between arrays

```
In [3]: A = X + Y             # element-wise addition
        M = X * Y             # element-wise multiplication
        D = numpy.dot(X, Y)   # dot product
        T = X.T               # transposing
        X_tail = X[2:]        # indexing (similar to lists)

        A, M, D, T, X_tail

Out[3]: (array([ 6,  8, 10, 12]),
         array([ 5, 12, 21, 32]),
         70,
         array([1, 2, 3, 4]),
         array([3, 4]))
```

```
In [4]: # Compare this to operations on lists
        X_list = [1, 2, 3, 4]
        Y_list = [5, 6, 7, 8]
        print(X_list + Y_list)
        print(X_list * Y_list) #  -> raises Exception
```

```
[1, 2, 3, 4, 5, 6, 7, 8]



        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-4-aa39a4918754> in <module>()
          3 Y_list = [5, 6, 7, 8]
          4 print(X_list + Y_list)
    ----> 5 print(X_list * Y_list) #  -> raises Exception


        TypeError: can't multiply sequence by non-int of type 'list'
```

Equivalent operations with lists

```
In [5]: A_list = [x + y for x, y in zip(X, Y)]      # element-wise addition
        M_list = [x * y for x, y in zip(X, Y)]      # element-wise multiplication
        D_list = sum([x * y for x, y in zip(X, Y)]) # dot product

        A_list, M_list, D_list
```

```
Out[5]: ([6, 8, 10, 12], [5, 12, 21, 32], 70)
```

Observation: Results are the same, but the Numpy syntax is much more readable (i.e. more compact) than the Python syntax for the same vector operations.

Shapes of arrays

```
In [6]: print(A.shape, M.shape, D.shape)
```

```
(4,) (4,) ()
```

Matrices

```
In [7]: A = numpy.array(
            [
                [1, 2, 3],
                [4, 5, 6]
            ]
        )
```

```
In [8]: print(A)
        print(10 * "-")
        print(A.shape)
```

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

```
In [9]: # Elementwise multiplication
        A * A
```

```
Out[9]: array([[ 1,  4,  9],
               [16, 25, 36]])
```

```
In [10]: # Matrix-matrix multiplication
         numpy.dot(A, A.T)
```

```
Out[10]: array([[14, 32],
                [32, 77]])
```

Observation: Unlike Matlab, "*" denotes an element-wise multiplication. Matrix multiplication is instead implemented by the function "dot".

```
In [11]: numpy.dot(A, A)   # -> raises Exception
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-11-832ddb5ea8b3> in <module>()
----> 1 numpy.dot(A, A)   # -> raises Exception


ValueError: shapes (2,3) and (2,3) not aligned: 3 (dim 1) != 2 (dim 0)
```

Performance evaluation

To verify that in addition to the more compact syntax, Numpy also provides a computational benefit over standard Python, we compare the running time of a similar computation performed in pure Python and in Numpy. The module "time" provides a function "clock" to measure the current time.

```
In [12]: import time
         time.clock()
```

```
Out[12]: 2.468887
```

we now wait a little bit...

```
In [13]: time.clock()
```

```
Out[13]: 2.488098
```

and can observed that the value is higher than before (time has passed). We now define two functions to test the speed of matrix multiplication for two $n \times n$ matrices.

```
In [14]: # pure Python implementation

         def benchmark_python(n):

             # initialization
             X = numpy.ones((n, n))
             Y = numpy.ones((n, n))
             Z = numpy.zeros((n, n))

             # actual matrix multiplication
             start = time.clock()
             for i in range(n):
                 for j in range(n):
                     for k in range(n):
                         Z[i,j] += X[i, k] * Y[k, j]
             end = time.clock()

             return end-start
```

```
In [15]: # Numpy implementation

         def benchmark_numpy(n):

             # initialization
```

```
    X = numpy.ones((n, n))
    Y = numpy.ones((n, n))
    Z = numpy.zeros((n, n))

    # actual matrix multiplication
    start = time.clock()
    Z = numpy.dot(X, Y)
    end = time.clock()

    return end-start
```

Evaluating this function for $n = 100$ iterations, we can observe that Numpy is much faster than pure Python.

```
In [16]: num_iterations = 100
         benchmark_python(num_iterations), benchmark_numpy(num_iterations)
```

```
Out[16]: (1.5302230000000003, 0.004741000000000106)
```

```
In [17]: ### Common alternative way of importing Numpy: alias "np"
         import numpy as np
         print(np.ones((3, 3)))
```

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

Plotting

In machine learning, it is often necessary to visualize the data, or to plot properties of algorithms such as their accuracy or their speed. For this, we can make use of the matplotlib library, which we load with the following sequence of commands.

```
In [18]: import matplotlib
         import matplotlib.pyplot as plt
         # Needed in Jupyter Notebook
         %matplotlib inline
```

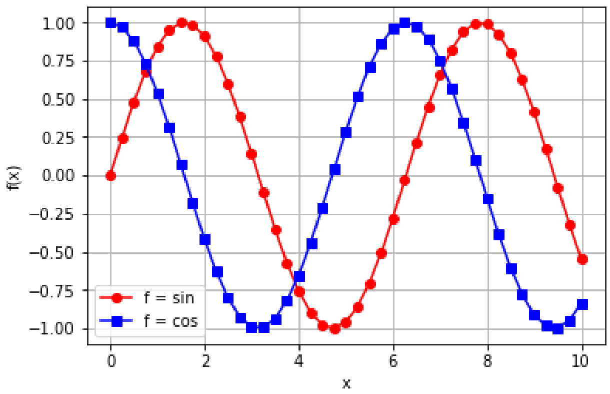
Basic plot

```
In [19]: x = numpy.arange(0, 10.001, 0.25)
         y = numpy.sin(x)
         z = numpy.cos(x)

         plt.plot(x, y, 'o-', color='red', label='f = sin')
         plt.plot(x, z, 's-', color='blue', label='f = cos')

         plt.legend()

         plt.xlabel('x')
         plt.ylabel('f(x)')
         plt.grid(True)
```



Plotting a performance curve for matrix multiplication
We run the computation with different parameters (e.g. size of input arrays)

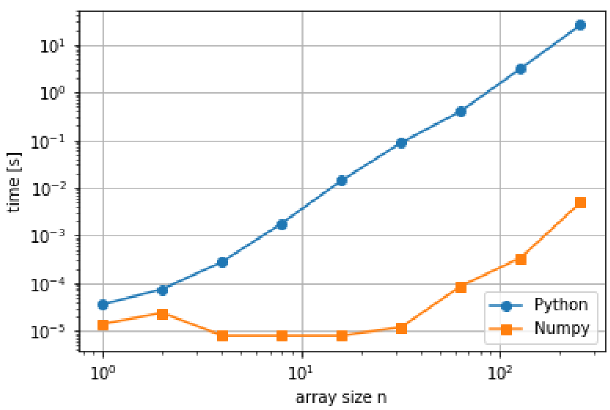
```
In [20]: N = [1, 2, 4, 8, 16, 32, 64, 128, 256]

         py_t = [benchmark_python(n) for n in N]
         np_t = [benchmark_numpy(n) for n in N]
```

Then, we render the plot

```
In [21]: plt.plot(N, py_t, 'o-', label='Python')
         plt.plot(N, np_t, 's-', label='Numpy')
         plt.grid(True)
         plt.xscale('log')
         plt.yscale('log')
         plt.xlabel('array size n')
         plt.ylabel('time [s]')
         plt.legend(loc='lower right')

Out[21]: <matplotlib.legend.Legend at 0x7f98803567f0>
```



Advanced Numpy

Special Array Initializations

Numpy arrays can be initialized to specific values (numpy.zeros, numpy.ones, ...). Special numpy arrays (e.g. diagonal, identity) can be created easily.

```
In [25]: A = numpy.zeros((3, 3))          # array of size 2x2 filled with zeros
         B = numpy.ones((3, 3))           # same, but filled with ones
         C = numpy.diag((1.0, 2.0, 3.0)) # diagonal matrix
         D = numpy.eye(3)                 # identity matrix
         E = numpy.random.rand(3, 3)      # random numbers
         F = numpy.triu(B)                # upper triagonal matrix

         print(A)
         print(B)
         print(C)
         print(D)
         print(E)
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
[[ 1.  0.  0.]
 [ 0.  2.  0.]
 [ 0.  0.  3.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
[[ 0.45292187  0.9226538   0.14548827]
 [ 0.97963672  0.36689856  0.81445493]
 [ 0.07162647  0.83952406  0.28105698]]
```

Array type

```
In [26]: A = numpy.ones((2, 2))
         type(A), A.shape, A.size, A.ndim, A.dtype
```

```
Out[26]: (numpy.ndarray, (2, 2), 4, 2, dtype('float64'))
```

```
In [27]: A = numpy.ones((3,3,3))
         type(A), A.shape, A.size, A.ndim, A.dtype
```

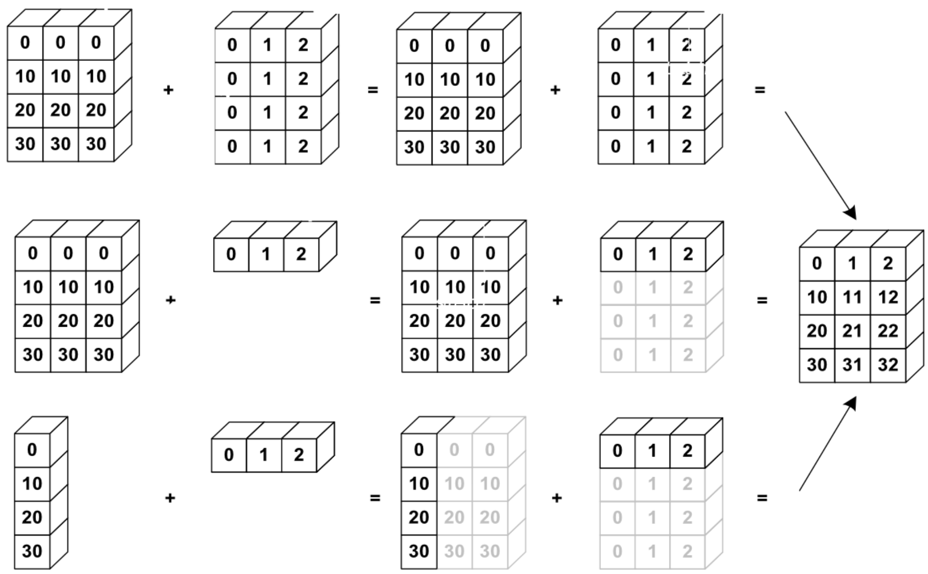
```
Out[27]: (numpy.ndarray, (3, 3, 3), 27, 3, dtype('float64'))
```

Casting

An array can be explicitly forced to have elements of a certain type (e.g. half-precision). When applying an operator to two arrays of different types, the returned array retains the type of the highest-precision input array (here, float64).

```
In [28]: E = A.astype('float32')
         A.dtype, E.dtype, (A + E).dtype
```

```
Out[28]: (dtype('float64'), dtype('float32'), dtype('float64'))
```



Numpy broadcasting

Reshaping and transposing

```
In [29]: A = numpy.array([[1, 2, 3], [4, 5, 6]])

         print(A)
         print(A.reshape((3,2)))
         print(A.ravel())
         print(A.T)
```

```
[[1 2 3]
 [4 5 6]]
[[1 2]
 [3 4]
 [5 6]]
[1 2 3 4 5 6]
[[1 4]
 [2 5]
 [3 6]]
```

Broadcasting

See also https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

```
In [30]: numpy.ones((3, 2)) + 1
```

```
Out[30]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

```
In [31]: numpy.ones((3, 2)) + numpy.ones((3, 2))
```

```
Out[31]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

```
In [32]: numpy.ones((3, 1)) + numpy.ones((1, 2))
```

```
Out[32]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

```
In [33]: numpy.ones((3, 1)) + numpy.ones((2))
```

```
Out[33]: array([[ 2.,  2.],
                [ 2.,  2.],
                [ 2.,  2.]])
```

Indexing

See also https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

```
In [34]: A = numpy.arange(30).reshape(6, 5)
         print(A)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]
 [25 26 27 28 29]]
```

Select rows/columns

```
In [35]: print(A[3, :])
         print(A[:, 3])
```

```
[15 16 17 18 19]
[ 3  8 13 18 23 28]
```

Select window

```
In [36]: print(A[1:5, 1:4])
```

```
[[ 6  7  8]
 [11 12 13]
 [16 17 18]
 [21 22 23]]
```

Select even rows and odd columns

```
In [37]: print(A[::2, 1::2])
```

```
[[ 1  3]
 [11 13]
 [21 23]]
```

Select last two columns

```
In [38]: print(A[:, -2:])
```

```
[[ 3  4]
 [ 8  9]
 [13 14]
 [18 19]
 [23 24]
 [28 29]]
```

Select column 1 and 4

```
In [39]: print(A[:, [1, 4]])
```

```
[[ 1  4]
 [ 6  9]
 [11 14]
 [16 19]
 [21 24]
 [26 29]]
```

Boolean Arrays

```
In [40]: a = numpy.random.rand(4, 4)
         print(a)
         b = a > 0.5
         print(b)
         print(b.astype(int))
         print(a[b])
```

```
[[ 0.08718976  0.40300868  0.657437    0.44630721]
 [ 0.69862314  0.12675595  0.04560769  0.78445988]
 [ 0.76957838  0.34242874  0.81048327  0.03781725]
 [ 0.47534253  0.42109713  0.78646799  0.8152729 ]]
[[False False  True False]
 [ True False False  True]
 [ True False  True False]
 [False False  True  True]]
[[0 0 1 0]
 [1 0 0 1]
 [1 0 1 0]
 [0 0 1 1]]
[ 0.657437    0.69862314  0.78445988  0.76957838  0.81048327  0.78646799
  0.8152729 ]
```

```
In [41]: # Is any/all of the elements True?
         numpy.any(b), numpy.all(b)
```

```
Out[41]: (True, False)
```

```
In [42]: # Apply to specific axes only
         numpy.any(b, axis=1), numpy.all(b, axis=0)
```

```
Out[42]: (array([ True,  True,  True,  True], dtype=bool),
          array([False, False, False, False], dtype=bool))
```

Analyzing a Dataset

Let's load the Boston dataset (506 examples composed of 13 features each).

In [43]:
```python
# extract two interesting features of the data
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.keys())

X = boston['data']
F = boston['feature_names']

print(F)
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR'])
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

Reduce-type operations

In [44]:
```python
print(X.mean())                                 # Global dataset mean feature value
print(X[:, 0].mean())                           # Mean of first feature (CRIM)
print(X.mean(axis=0), X.mean(axis=0).shape)     # Mean of all features
print(X.std(axis=0), X.std(axis=0).shape)       # Standard deviation of all features
print(X.min(), X.max())                         # Extreme values
print(X.shape, X.sum(axis=1).shape, X.sum(axis=1, keepdims=True).shape)
```

```
70.0724468258
3.59376071146
[  3.59376071e+00   1.13636364e+01   1.11367787e+01   6.91699605e-02
   5.54695059e-01   6.28463439e+00   6.85749012e+01   3.79504269e+00
   9.54940711e+00   4.08237154e+02   1.84555336e+01   3.56674032e+02
   1.26530632e+01] (13,)
[  8.58828355e+00   2.32993957e+01   6.85357058e+00   2.53742935e-01
   1.15763115e-01   7.01922514e-01   2.81210326e+01   2.10362836e+00
   8.69865112e+00   1.68370495e+02   2.16280519e+00   9.12046075e+01
   7.13400164e+00] (13,)
0.0 711.0
(506, 13) (506,) (506, 1)
```

In [45]:
```python
# Show the feature name along with the mean and standard deviation
list(zip(F, X.mean(axis=0), X.std(axis=0)))
```

Out[45]:
```
[('CRIM', 3.5937607114624508, 8.5882835476535533),
 ('ZN', 11.363636363636363, 23.299395694766027),
 ('INDUS', 11.136778656126504, 6.8535705833908729),
 ('CHAS', 0.069169960474308304, 0.25374293496034855),
 ('NOX', 0.55469505928853724, 0.11576311540656153),
 ('RM', 6.2846343873517867, 0.7019225143345692),
 ('AGE', 68.574901185770784, 28.121032570236885),
 ('DIS', 3.795042687747034, 2.1036283563444589),
 ('RAD', 9.5494071146245059, 8.6986511177906447),
 ('TAX', 408.23715415019763, 168.37049503938141),
 ('PTRATIO', 18.455533596837967, 2.1628051914821418),
 ('B', 356.67403162055257, 91.204607452172723),
 ('LSTAT', 12.653063241106723, 7.1340016366504848)]
```

Retain two interesting features (5 and 12)

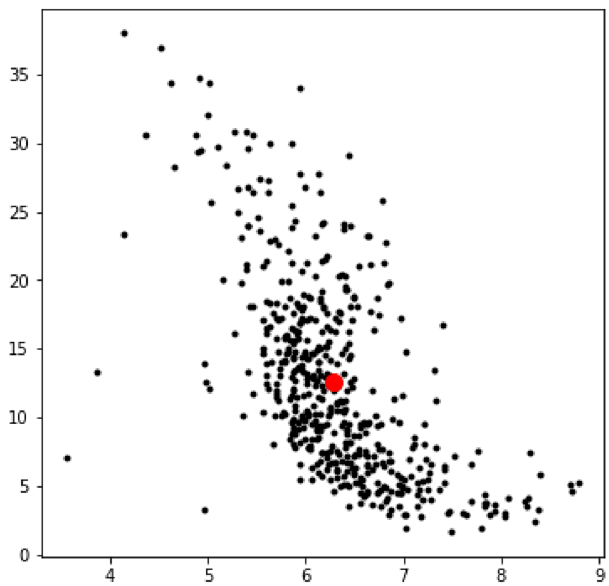
```
In [46]: X = X[:, [5, 12]]
         print(X.shape)
```

```
(506, 2)
```

Scatter-plot the first two dimensions

```
In [47]: plt.figure(figsize=(6, 6))
         plt.plot(X[:, 0], X[:, 1], 'o', color='black', ms=3)
         plt.plot(X[:, 0].mean(), X[:, 1].mean(), 'o', color='red', ms=10)
```

```
Out[47]: [<matplotlib.lines.Line2D at 0x7fb9516a0550>]
```

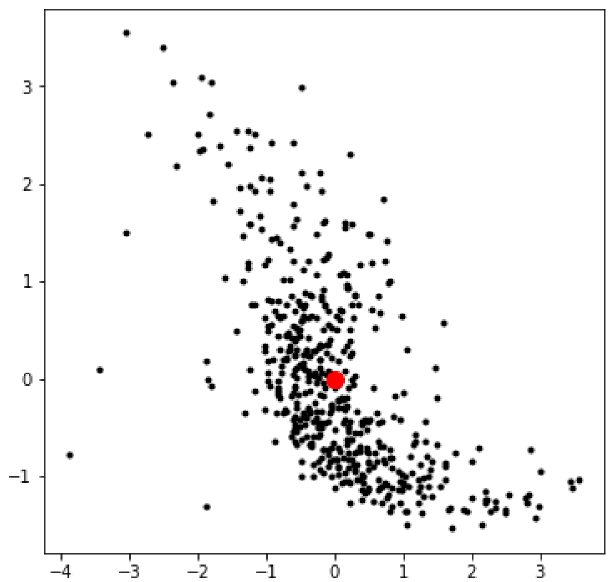


Normalize the data

```
In [48]: X_norm = X - X.mean(axis=0) # center around origin
         X_norm /= X.std(axis=0)  # rescale features so that they have standard deviation 1 in each dim
```

```
In [49]: plt.figure(figsize=(6, 6))
         plt.plot(X_norm[:, 0], X_norm[:, 1], 'o', color='black', ms=3)
         plt.plot(X_norm[:, 0].mean(), X_norm[:, 1].mean(), 'o', color='red', ms=10)
```

```
Out[49]: [<matplotlib.lines.Line2D at 0x7fb9516334e0>]
```



Computing a distance matrix

```
In [50]: import scipy
         import scipy.spatial

         D = scipy.spatial.distance.cdist(X_norm, X_norm)
```

alternative way of computing a distance matrix:

```
In [51]: Dalt = np.sqrt(((X_norm ** 2).sum(axis=1).reshape((1, len(X_norm))) \
               + (X_norm ** 2).sum(axis=1).reshape((len(X_norm), 1)) \
               - 2 * numpy.dot(X_norm, X_norm.T)) + 1e-7)

         print(((Dalt - D) ** 2).mean())

1.97641741612e-10
```

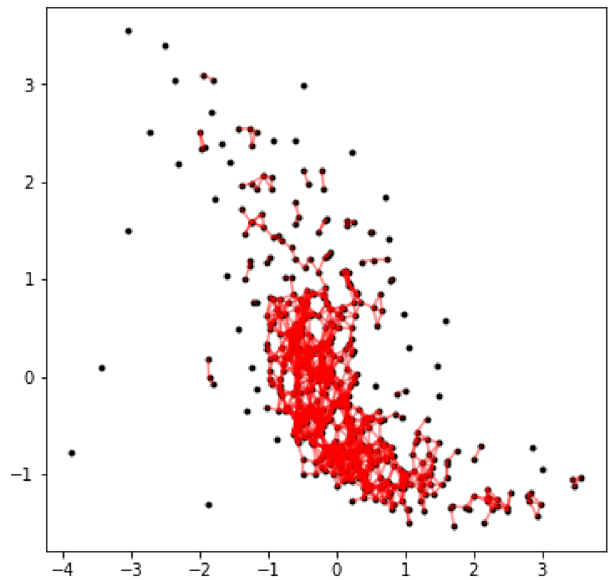
Highlighting nearby data points

```
In [52]: plt.figure(figsize=(6, 6))

         ind = numpy.where(D < 0.2)

         plt.plot(X_norm[:, 0], X_norm[:, 1], 'o', color='black', ms=3)

         for i1,i2 in zip(*ind):
             plt.plot([X_norm[i1, 0], X_norm[i2, 0]], [X_norm[i1, 1], X_norm[i2, 1]], color='red', alpha
```



Getting help

In [53]: help(numpy.where)

```
Help on built-in function where in module numpy.core.multiarray:

where(...)
    where(condition, [x, y])

    Return elements, either from `x` or `y`, depending on `condition`.

    If only `condition` is given, return ``condition.nonzero()``.

    Parameters
    ----------
    condition : array_like, bool
        When True, yield `x`, otherwise yield `y`.
    x, y : array_like, optional
        Values from which to choose. `x` and `y` need to have the same
        shape as `condition`.

    Returns
    -------
    out : ndarray or tuple of ndarrays
        If both `x` and `y` are specified, the output array contains
        elements of `x` where `condition` is True, and elements from
        `y` elsewhere.

        If only `condition` is given, return the tuple
        ``condition.nonzero()``, the indices where `condition` is True.

    See Also
```

```
--------
nonzero, choose

Notes
-----
If `x` and `y` are given and input arrays are 1-D, `where` is
equivalent to::

    [xv if c else yv for (c,xv,yv) in zip(condition,x,y)]

Examples
--------
>>> np.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...          [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])

>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))

>>> x = np.arange(9.).reshape(3, 3)
>>> np.where( x > 5 )
(array([2, 2, 2]), array([0, 1, 2]))
>>> x[np.where( x > 3.0 )]               # Note: result is 1D.
array([ 4.,  5.,  6.,  7.,  8.])
>>> np.where(x < 5, x, -1)               # Note: broadcasting.
array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])

Find the indices of elements of `x` that are in `goodvalues`.

>>> goodvalues = [3, 4, 7]
>>> ix = np.in1d(x.ravel(), goodvalues).reshape(x.shape)
>>> ix
array([[False, False, False],
       [ True,  True, False],
       [False,  True, False]], dtype=bool)
>>> np.where(ix)
(array([1, 1, 2]), array([0, 1, 1]))
```

In [ ]: