Python for scientific computing

Python has extensive packages to help with data analysis:

- numpy: matrices, linear algebra, Fourier transform, pseudorandom number generators
- scipy: advanced linear algebra and maths, signal processing, statistics
- pandas: DataFrames, data wrangling and analysis
- matplotlib: visualizations such as line charts, histograms, scatter plots.

```
<IPython.core.display.HTML object>
```

NumPy

NumPy is the fundamental package required for high performance scientific computing in Python. It provides:

- `ndarray`: fast and space-efficient n-dimensional numeric array with vectorized arithmetic operations
- Functions for fast operations on arrays without having to write loops
- Linear algebra, random number generation, Fourier transform
- Integrating code written in C, C++, and Fortran (for faster operations)

`pandas` provides a richer, simpler interface to many operations. We'll focus on using `ndarrays` here because they are heavily used in scikit-learn.

ndarrays

There are several ways to create numpy arrays.

```
[74]: # Convert normal Python array to 1-dimensional numpy array
      np.array((1, 2, 53))
```

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

```
[75]: # Convert sequences of sequences of sequences ... to n-dim array
      np.array([(1.5, 2, 3), (4, 5, 6)])
```

```
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

```
[76]: # Define element type at creation time
      np.array([[1, 2], [3, 4]], dtype=complex)
```

```
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Useful properties of ndarrays:

[77]:
```
my_array = np.array([[1, 0, 3], [0, 1, 2]])
my_array.ndim     # number of dimensions (axes), also called the rank
my_array.shape    # a matrix with n rows and m columns has shape (n,m)
my_array.size     # the total number of elements of the array
my_array.dtype    # type of the elements in the array
my_array.itemsize # the size in bytes of each element of the array
```

```
2
```

```
(2, 3)
```

```
6
```

```
dtype('int64')
```

```
8
```

Quick array creation.
It is cheaper to create an array with placeholders than extending it later.

[78]:
```
np.ones(3) # Default type is float64
np.zeros([2, 2])
np.empty([2, 2]) # Fills the array with whatever sits in memory
np.random.random((2,3))
np.random.randint(5, size=(2, 4))
```

```
array([ 1.,  1.,  1.])
```

```
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
array([[ 0.681,  0.545,  0.669],
       [ 0.181,  0.47 ,  0.682]])
```

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

Create sequences of numbers

[79]:
```
np.linspace(0, 1, num=4)   # Linearly distributed numbers between 0 and 1
np.arange(0, 1, step=0.3)  # Fixed step size
np.arange(12).reshape(3,4) # Create and reshape
np.eye(4)                  # Identity matrix
```

```
array([ 0.   ,  0.333,  0.667,  1.   ])
```

```
array([ 0. ,  0.3,  0.6,  0.9])
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Basic Operations

Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result. Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

```
[80]: a = np.array([20, 30, 40, 50])
      b = np.arange(4)
      a, b    # Just printing
      a-b
      b**2
      a > 32
      a += 1
      a
```

```
(array([20, 30, 40, 50]), array([0, 1, 2, 3]))
```

```
array([20, 29, 38, 47])
```

```
array([0, 1, 4, 9])
```

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

```
array([21, 31, 41, 51])
```

The product operator * operates elementwise.
The matrix product can be performed using dot()

```
[81]: A, B = np.array([[1,1], [0,1]]), np.array([[2,0], [3,4]]) # assign multiple variables i
      A
      B
      A * B
      np.dot(A, B)
```