**Lecture 11 (week 6.2)**

Thursday October 13th 2016

This lecture wraps up our description of Python containers with dictionaries. The rest of the lecture will be our first look at **scipy**.

Dictionaries

Python 's dictionaries are hash tables (<https://en.wikipedia.org/wiki/Hash_table>), worded simply, it is a mapping between a collections of objects. They work like **associative arrays** or hashes and consist of *key-value* pairs. Keys can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces **{ }** and values can be assigned and accessed using square braces **[ ]**.

Here are some basic creation, deletion and visualization tools for dictionaries:

**In [3]: a = {'eyecolour':'blue','height':152.0,42:'the answer'}**

**In [4]: a**

**Out[4]: {'height': 152.0, 'eyecolour': 'blue', 42: 'the answer'}**

**In [5]: a['age']=28**

**In [6]: a**

**Out[6]: {'height': 152.0, 'eyecolour': 'blue', 42: 'the answer', 'age': 28}**

**In [7]: del(a['height'])**

**In [8]: a**

**Out[8]: {'eyecolour': 'blue', 42: 'the answer', 'age': 28}**

**In [9]: a.keys()**

**Out[9]: dict\_keys(['eyecolour', 42, 'age'])**

**In [10]: a.values()**

**Out[10]: dict\_values(['blue', 'the answer', 28])**

**In [11]: b={}** *# empty dictionary*

**In [13]: type(b)**

**Out[13]: dict**

**In [14]: b['hello']='Hi !'**

**In [15]: b**

**Out[15]: {'hello': 'Hi !'}**

Here is a list of dictionary methods:

[dict.clear()](http://www.tutorialspoint.com/python/dictionary_clear.htm)  
Removes all elements of dictionary *dict*

[dict.copy()](http://www.tutorialspoint.com/python/dictionary_copy.htm)  
Returns a shallow copy of dictionary *dict*

[dict.fromkeys()](http://www.tutorialspoint.com/python/dictionary_fromkeys.htm)  
Create a new dictionary with keys from seq and values *set* to *value*.

[dict.get(key, default=None)](http://www.tutorialspoint.com/python/dictionary_get.htm)  
For *key* key, returns value or default if key not in dictionary

[dict.has\_key(key)](http://www.tutorialspoint.com/python/dictionary_has_key.htm)  
Returns *true* if key in dictionary *dict*, *false* otherwise

[dict.items()](http://www.tutorialspoint.com/python/dictionary_items.htm)  
Returns a list of *dict*'s (key, value) tuple pairs

[dict.keys()](http://www.tutorialspoint.com/python/dictionary_keys.htm)  
Returns list of dictionary dict's keys

[dict.setdefault(key, default=None)](http://www.tutorialspoint.com/python/dictionary_setdefault.htm)  
Similar to get(), but will set dict[key]=default if *key* is not already in dict

[dict.update(dict2)](http://www.tutorialspoint.com/python/dictionary_update.htm)  
Adds dictionary *dict2*'s key-values pairs to *dict*

[dict.values()](http://www.tutorialspoint.com/python/dictionary_values.htm)  
Returns list of dictionary *dict2*'s values

Introduction to SciPy

Python SciPy is a powerful library for scientific computing. It is **NOT** a symbolic mathematical solver (unlike Mathematica), although there is a symbolic calculation library still in development called SymPy (<http://www.sympy.org/en/index.html>). SciPy contains packages that can handle your numerical tasks, form data analysis to linear algebra and differential equation solving. The Wikipedia page gives you a general description of the library <https://en.wikipedia.org/wiki/SciPy>.

The full SciPy documentation can be found there:

<http://docs.scipy.org/doc/scipy/reference/index.html>

This link starts with tutorials focused on some of the most common numerical tasks, followed by the full list of APIs (Application Programming Interface). From this list you have already used **curve\_fit** located in the **scipy.optimize** package. In the coming weeks we will explore a few more, those which will be useful for your projects. More specifically:

linear algebra (in the **linalg** package)

integration and ODEs (from the **integrate** package)

root finding (in the **optimize** package)

When you click on one of the APIs links, you are presented with a list of functions (methods), when you lick on one of these you are presented with a description what the function do, how to call it (its arguments optional or not), what it returns and usually (but not always) an example of how to use it.

It is common practice to import the packages one needs from scipy instead of importing all of scipy, e.g.:

**from scipy import linalg**

Note that *some* functionalities in *some* SciPy packages are redundant with existing features of other Python libraries, e.g. in numpy. This is for historical reason and in general redundant fonctionalities can still be used. This is the case for the matrix calculations tools in numpy which are also available from the linear algebra package **linalg** in SciPy. It is also the case for Fourier Transforms, both scipy and numpy have **fft** packages. It is recommended to use the SciPy packages when available. However, some of the numpy matrices operations are still being widely used, so I will review them today and show you how you can achieve the same result with **linalg**.

Array versus matrix calculation

Matrix calculations can be done with numpy and scipy, but it is important to remember that array and matrix objects and operations are not the same thing!

The following shows you how to define a matrix from a numpy array and some of the most basic operations you can do:

**In [1]: import numpy as np ; from scipy import linalg**

**In [2]: a = np.array([[1,2],[3,4]])**

**In [3]: m = np.mat(a)** *# convert 2-d array to matrix*

**In [4]: m**

**Out[4]:**

**matrix([[1, 2],**

**[3, 4]])**

**In [5]: a[0]** *# result is 1-dimensional*

**Out[5]: array([1, 2])**

**In [6]: m[0]** *# result is 2-dimensional*

**Out[6]: matrix([[1, 2]])**

**In [7]: a\*a** *# element-by-element multiplication*

**Out[7]:**

**array([[ 1, 4],**

**[ 9, 16]])**

**In [8]: m\*m** *# (algebraic) matrix multiplication*

**Out[8]:**

**matrix([[ 7, 10],**

**[15, 22]])**

**In [9]: a\*\*3** *# element-wise power*

**Out[9]:**

**array([[ 1, 8],**

**[27, 64]])**

**In [10]: m\*\*3** *# matrix multiplication m\*m\*m*

**Out[10]:**

**matrix([[ 37, 54],**

**[ 81, 118]])**

**In [11]: m.T** *# transpose of the matrix*

**Out[11]:**

**matrix([[1, 3],**

**[2, 4]])**

**In [12]: m.H** *# conjugate transpose (differs from .T for complex matrices)*

**Out[12]:**

**matrix([[1, 3],**

**[2, 4]])**

**In [13]: m.I** *# inverse matrix*

**Out[13]:**

**matrix([[-2. , 1. ],**

**[ 1.5, -0.5]])**

**In [41]: linalg.inv(a)** *# same as above using scipy.linalg*

**Out[41]:**

**array([[-2. , 1. ],**

**[ 1.5, -0.5]])**

**In [42]: linalg.det(a)** *# determinant calculation*

**Out[42]: -2.0**

The full list of attributes and methods of the matrix package is given there:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html>

The full list of functionalities from the linear algebra package **linalg** from scipy is there:

<http://docs.scipy.org/doc/scipy/reference/linalg.html>

and from **numpy.linalg**:

<http://docs.scipy.org/doc/numpy/reference/routines.linalg.html>

Numpy array broadcasting

Numpy arrays operations can be performed even when arrays don’t have the same shape, this is the concept of broadcasting I briefly mentioned before. It is useful to know how arrays of different shapes can be combined together. I want you to learn about it by yourself on this link, scroll down to Broadcasting (you saw before):

<http://cs231n.github.io/python-numpy-tutorial/#numpy>