**Lecture 12 (week 7.1)**

Tuesday October 18th 2016

Miscellaneous

This lecture addresses various loose ends sometimes briefly mentioned in class, sometime totally overlooked. You will find some useful tips and material in here. It is presented in random order.

Keywords and built-in functions

Some keywords and built-in functions are really useful and we have covered only less than half of them. Take some time to explore the others and thing how/if they can be useful for your projects.

Python3 programming keywords:

<http://www.programiz.com/python-programming/keyword-list>

Python3 built-in functions:

<https://docs.python.org/3/library/functions.html>

Of particular interests are **lambda**, **isinstance()**, **zip()**

The anonymous function creator **lambda** is particularly useful:

<http://www.secnetix.de/olli/Python/lambda_functions.hawk>

The keyword **class** is central to the Python Object Oriented Programming which is beyond the scope of this course. I may introduce **class** later in november, but you don’t need it for your projects.

Exception handling

We have not spend much time in debugging and error handling. For beginners, googling an error message usually gives you the solution to your error. There are however tools for error handling which can make your debugging task much easier as well as control the flow of execution to anticipate possible error. To this end, read the following:

<https://docs.python.org/3/tutorial/errors.html>

Arithmetic precision

We also haven’t said anything about numerical calculation precision. This becomes an important topic when you are doing serious computational work. You don’t want your scientific results to be bogus! Here is a section on this topic in the context of Python:

<https://docs.python.org/3/tutorial/floatingpoint.html>

A thorough discussion on this topic is given there:

<https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/02Numerics/Double/paper.pdf>

In place and not in place operation

In place or not in place function return: you always return something but whether you get a copy or you overwrite the old array depends on how you write the function. See below and it should be clear:

**#!/usr/bin/env python3**

**import numpy as np**

**import numpy.random as nr**

**def my\_sign\_orig(a):**

**a[a > 0] = 1**

**a[a < 0] = -1**

**a[a == 0] = 0**

**return a**

**def my\_sign\_copy(a):**

**b = a.copy()**

**b[a > 0] = 1**

**b[a < 0] = -1**

**b[a == 0] = 0**

**return b**

**a = nr.randint(-10,10,20)**

**b = my\_sign\_orig(a)**

**print(a)**

**print(b)**

**c = nr.randint(-10,10,20)**

**d = my\_sign\_copy(c)**

**print(c)**

**print(d)**

Calling the Python interpreter

On a different computer, to make sure you call Python 3 and not Python 2, it is safer to start your interactive python session with **ipython3**. The call '**ipython3 -pylab**' does two things:

- It calls '**from pylab import \***' within ipython3.

* It enables a special ipython threading mode that allows you to use the interactive ipython interpreter and an active plotting window at the same time.
* It creates a way to call functionalities which very much looks like **MATLAB**.

The first point is that it imports everything from **pylab** to the global name space (that why it looks like **MATLAB**) and the second point solves the **plt.show()**-problem within an interactive ipython session.

<http://stackoverflow.com/questions/458209/is-there-a-way-to-detach-matplotlib-plots-so-that-the-computation-can-continue>

Running a Python session

If you have a preference for window/GUI environment over the command line interface, two of the most common options are **Spyder** or using the **ipython notebook**.

**Spyder** offers a nice combo text editing/python prompt/graphic window in one unique environment pretty much **MATLAB**-like, so you don’t have to use **nano** for editing or start **ipython** manually. It is nice, but don’t try to run it over the network (e.g. when working from home). It will be VERY SLOW as all the environment information will have to travel back and forth over internet between our PHAS machine and your home computer. You can try in class though, from the local machines. Here is a description of **Spyder**:

<https://pypi.python.org/pypi/spyder>

it is installed on the PHAS machines.

The other option is notebook. From the Linux prompt you type

**ipython notebook**

It will open an environment in your web browser which looks like **MATHEMATICA**. It is nice too, but you also don’t want to run this across the network for the same reason as above.

This link gives you more information about the notebook:

<https://ipython.org/ipython-doc/3/notebook/notebook.html#configuring-the-ipython-notebook>

Note that this is however becoming obsolete and the notebook feature has now been fully integrated in the **jupyter** environment which provides much more than the notebook feature, e.g. it has a full interface with other programing languages:

<http://jupyter.org/>

These GUI solutions are nice when you run **python** locally, otherwise stick to the command line based solution. if your X11 forwarding is broken or too slow.

Function vectorization

In assignment 9 you have worked out a way to make the function **my\_sign()** numpy compliant, i.e. when called with an array, it returns an array. The numpy method **vectorize()** can do this for you so you don’t have to rewrite a function to make it work explicitly for arrays.

**HOWEVER** I would not recommend using it for time consuming tasks which require optimized operations (e.g. the mandelbrot set exercises). The reason is that **vectorize()** actually performs a *fake* vectorization. Internally, it takes your function and applies it sequentially to every element of the input array, i.e. very much like what a **for** loop would do. As you know from assignment 9, the use of loops is far from being optimal for time critical tasks. So **vectorize()** can be used for small tasks only, otherwise, I would not use it. Here is how it works: we start with the un-optimized **my\_sign()** function:

**import numpy as np**

**def my\_sign(a):**

**if a > 0:**

**return 1**

**if a < 0:**

**return -1**

**if a == 0:**

**return 0**

When called with a numpy array you get this error:

**In [46]: my\_sign(a)**

**---------------------------------------------------------------------------**

**ValueError Traceback (most recent call last)**

**<ipython-input-46-66ff6a29971d> in <module>()**

**----> 1 my\_sign(a)**

**/Volumes/TEACHING/PHYS210/2016/script.py in my\_sign(a)**

**2**

**3 def my\_sign(a):**

**----> 4 if a > 0:**

**5 return 1**

**6 if a < 0:**

**ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()**

The method **vectorize(func)** allows you to "vectorize" the function **func()** which you can then use with arrays:

**In [47]: my\_sign\_vec=np.vectorize(my\_sign)**

**In [48]: my\_sign\_vec(a)**

**Out[48]:**

**array([ 1, -1, -1, 1, -1, 0, -1, 0, 1, -1, 0, -1, 1, -1, -1, -1, 1, 1, 1, -1])**

It works, but it is not optimal (see above).

Note that code optimization should not be your primary concern for now. It is more important to learn programming by writing clear and easy to read codes at the beginning because optimization can often result in a cryptic style.

Read an image

Reading an image in a numpy array is a useful thing to know how to do. It uses the miscellaneous routine **imread** from scipy. Below I’m reading an image from Vancouver located there (save it first):

<http://www.vancouvereconomic.com/wp-content/uploads/2015/05/Aerial-of-Vancouver-Downtown-web-1400x640.jpg>

**In [10]: from scipy.misc import imread, imsave, imresize**

**In [11]: a=imread('vancouver.jpg')**

**In [12]: type(a)**

**Out[12]: numpy.ndarray**

**In [13]: a[0,0]**

**Out[13]: array([188, 220, 233], dtype=uint8)**

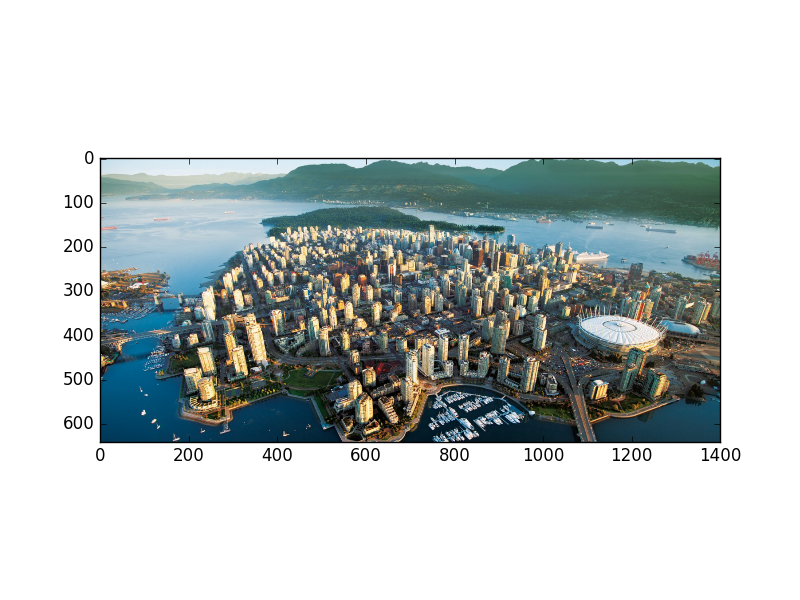
**In [15]: a.shape**

**Out[15]: (640, 1400, 3)**

**In [16]: import matplotlib.pyplot as plt**

**In [17]: plt.imshow(a)**

**Out[17]: <matplotlib.image.AxesImage at 0x1151891d0>**

**In [18]: plt.show()**

As you can see, the array a has 3 dimensions, this is so in order to support the RBG color system. Here are links on how to do this with the Python Imaging Library (PIL) package:

<http://stackoverflow.com/questions/10443295/combine-3-separate-numpy-arrays-to-an-rgb-image-in-python>

<http://www.pythonware.com/products/pil/>

Note that you can also read a WEB addresses as file names:

<http://stackoverflow.com/questions/7391945/how-do-i-read-image-data-from-a-url-in-python>

Animations

There is nothing better than learning by reverse engineering for this one. Look at some of these animations packages, the simple ones (e.g. the basic examples) and try to understand what they do. I will spend some time on this in a lecture later, animations will be nice for some of the projects.

<http://matplotlib.org/examples/animation/>

Note that on MAC OS, you have to change **blit=True** to **blit=False** for the animations to work.

Importing packages

The order by which you import packages **DOES** matter when there are functions with the same name in different packages. For instance we know that the **np.sqrt()** from numpy does not support with complex numbers while **scipy.sqrt()** does. The rule in Python is that for functions with same name in more than one package, **the latest import takes over**.

**In [7]: from numpy import \***

**In [8]: from scipy import \***

**In [9]: sqrt(-1)**

**Out[9]: 1j**

**In [10]: from scipy import \***

**In [11]: from numpy import \***

**In [12]: sqrt(-1)**

**/Users/waerbeke/anaconda3/bin/ipython:1: RuntimeWarning: invalid value encountered in sqrt**

**#!/bin/bash /Users/waerbeke/anaconda3/bin/python.app**

**Out[12]: nan**

Of course I warned you about using **import \*** before, but now that you know there are redundant functionalities in numpy and scipy libraries, you have to be extra careful with this, and the safest approach is to import only what you need.

Passing function names as arguments in functions

You can pass function names as arguments as shown in the example below:

**def doit(a, func, y, z):**

**result = z**

**result = func(a, y, result)**

**return result**

**def subfct1(arg1, arg2, arg3):**

**thing = (arg1 + arg2) / arg3**

**return thing**

**def subfct2(arg1, arg2, arg3):**

**thing = arg1 + (arg2 / arg3)**

**return thing**

Then run the following:

**In [13]: doit(1.,subfct1,2.,3.)**

**Out[13]: 1.0**

**In [14]: doit(1.,subfct2,2.,3.)**

**Out[14]: 1.6666666666666665**

Packing and unpacking arguments with **\*** and **\*\***

You have certainly already noticed, in some Python routines documents and examples, the use of **\*args** and **\*\*kwargs**. These are handy ways of passing a simple list of arguments for the former (as a list or a tuple), and a list of keys and values that will become a dictionary for the latter:

**\*args** = list of arguments -as positional arguments

**\*\*kwargs** = dictionary - whose keys become separate keyword arguments and the values become values of these arguments.

Note that **args** and **kwargs** are just names and can be named completely differently (although these names are the convention/norm).

Let’s explore with some examples. I first define these two functions:

**def fct1(\*\*kwargs):**

**print(type(kwargs),kwargs)**

**for name,value in kwargs.items():**

**print(name,value)**

**return**

**def fct2(\*args):**

**print(type(args),args)**

**for count,item in enumerate(args):**

**print('Object ',count,' is ',item,' its type is ',type(item))**

**return**

Then the following function calls will show you how to use the packing/unpacking of keyword/values pairs arguments in function calls, as you can see, this generates a dictionary:

**In [74]: fct1(key1='toto',key2=3.14)**

**<class 'dict'> {'key2': 3.14, 'key1': 'toto'}**

**key2 3.14**

**key1 toto**

**In [75]: fct1(key1='toto',key2=3.14,key3=[3.,4.])**

**<class 'dict'> {'key2': 3.14, 'key1': 'toto', 'key3': [3.0, 4.0]}**

**key2 3.14**

**key1 toto**

**key3 [3.0, 4.0]**

As you can see, you don’t have to worry how many arguments the function is called with, it is automatically taken care of by the **\*\*** operator.

Now with **fct2()** for a simple list of arguments:

**In [77]: fct2(1.1,2.3,'s')**

**<class 'tuple'> (1.1, 2.3, 's')**

**Object 0 is 1.1 its type is <class 'float'>**

**Object 1 is 2.3 its type is <class 'float'>**

**Object 2 is s its type is <class 'str'>**

**In [78]: fct2(1.1,2.3,'s',3.14159)**

**<class 'tuple'> (1.1, 2.3, 's', 3.14159)**

**Object 0 is 1.1 its type is <class 'float'>**

**Object 1 is 2.3 its type is <class 'float'>**

**Object 2 is s its type is <class 'str'>**

**Object 3 is 3.14159 its type is <class 'float'>**

**In [79]: b=1.1,2.3,'s',3.14159**

**In [80]: fct2(b)**

**<class 'tuple'> ((1.1, 2.3, 's', 3.14159),)**

**Object 0 is (1.1, 2.3, 's', 3.14159) its type is <class 'tuple'>**

**In [81]: fct2(\*b)**

**<class 'tuple'> (1.1, 2.3, 's', 3.14159)**

**Object 0 is 1.1 its type is <class 'float'>**

**Object 1 is 2.3 its type is <class 'float'>**

**Object 2 is s its type is <class 'str'>**

**Object 3 is 3.14159 its type is <class 'float'>**

Notice that the number of arguments can also vary. Look at the last two examples, using the **\*** packing operator in the function call allows you to pass the arguments as a tuple, which is automatically unpacked in the function.