Using Idrisi with Python:

An Introduction to the idrtools Package

By Andrew Shatz

***Introduction***

The material and exercises within this document will allow Python users of all levels to explore unique features and tools offered within idrtools including the analysis of raster and vector data using built-in Idrisi modules, the batch processing and analysis of large datasets, and the creation and implementation of analytical tools. The contents of this document are arranged as follows:

**Chapter 1: An Introduction to Python**

This chapter is meant for beginners and outlines the fundamentals of the Python language. Topics include variables, statements, elementary data types (integers, floats, strings, Booleans, etc), advanced data types (lists, tuples, dictionaries, etc), logical operators (if, elif, else, and, or, etc.), loops (for and while), functions, classes, and text file manipulation.

**Chapter 2: Exploring the idrtools User Interface (UI)**

This chapter will provide users with an introduction to the idrtools UI. Topics include managing projects, adding/editing/removing working and resource directories, organizing files, reading and writing Idrisi-specific file types, and updating file metadata information.

**Chapter 3: Using Modules for Data Display, Analysis and Batch Processing**

This chapter will provide users with an introduction to Idrisi module functionality within idrtools. Topics include data visualization and display, module use for batch processing, and module use for data analysis and preliminary tool building.

**Chapter 4: Idrisi Tool Development with “mytools”**

This chapter will provide the users with a demonstration of how to develop their own Idrisi tools using an idrtools framework. Topics include using developing a tool for the Area Under the Receiver Operating Characteristic curve (AUC), a validation method commonly used for Species Distribution Modeling (SDM).

***Chapter 1: An Introduction to Python***

*Sections*

Variables, Statements, and Data Types

Logical Operators

Advanced Data Types

Loops and Iterators

Functions and Classes

Text File Manipulation and Exception Handling

Tutorial under development…please stand by...

***Chapter 2: Exploring the idrtools User Interface (UI)***

*Sections*

The idrtools Environment

Working with Projects and Directories, and Files

Organizing Files and File Types

Reading and Writing Idrisi-specific File Types

Reading and Updating File Metadata

The data for this section may be found in a directory ending with the file path ‘\IDRISI Tutorial\Using IDRISI’ and \IDRISI Tutorial\Introductory GIS If the installation of the IDRISI program was set to default, these directory will be on the same drive as the installed IDRISI program.

***Section 2-1: The idrtools Environment***

Now that you have begun this chapter, you have at least a basic confidence in Python programing fundamentals! Now it is time to explore how idrtools uses the available tools from Python to coordinate and streamline analytical processes within the IDRISI Selva software.

To begin using idrtools, start IDRISI Selva by double-clicking on the IDRISI Selva icon. This package works in direct contact with the IDRISI API via a COM server, thus IDRISI must be open for idrtools to work. Then open the Python shell of your choosing (PythonWin or IDLE). Note that any screenshots of Python code in the remainder of this tutorial will be from IDLE. In the prompt, simply enter the following line below:

>>> from idrtools import \*

This single line indicates that everything within the initializer module for the idrtools package (idrtools/\_\_init\_\_.py) will be activated. Prior to the development of this package, there was an entire sequence of code needed to access the Idrisi API. This code looked a little bit something like the lines you see below:

>>> import win32com.client

>>> idrisi = win32com.client.Dispatch(“idrisi32.IdrisiAPIServer”)

With this code, users could only access the methods found within the Idrisi API via the COM server. The execution of the same processes is made much simpler using the code in step 3 above. Please note that prior to using this package, must have a working copy of the pywin32 package. The idrtools package would not function without it.

Now that idrtools is active within the shell, you may begin exploring some of the utilities available within this package. Please continue to the next section to learn about project and directory management.

***Section 2- 2: Working with Projects and Directories***

Now that you have activated the idrtools package within the Python shell, it is time to identify a project object (hey that rhymes). The project object is instantiated using the IdrisiExplorer class. But before we create our project object, we can identify all projects available on the PC through Idrisi Explorer using the built-in variable projects. Project files can be found in the ‘C:\Program Files (x86)\IDRISI Selva\Projects’ folder with ‘.env’ file extensions.

>>> projects

[‘default’]

Notice how the output of the projects variable is a list. All projects are stored as string variables within this list. In this particular case, there is only one project available called ‘default’, however this list is bound to grow with the addition of new projects. Now that we have a list of project files, it is time to create a project object using the code below:

>>> default = IdrisiExplorer(projects[0])

The object default is an instantiation of the IdrisiExplorer() class using the project called ‘default’,or ‘default.env’, but keep in mind that the file extension is not necessary within the class input. With the project object now active, we can examine the contents of the project. First let us look at the directories linked to the project.

>>> default.getProjectDirs()

['C:\\Users\\Public\\Documents\\IDRISI Tutorial Data\\Using Idrisi\\']

>>> default.getWorkingDir()

'C:\\Users\\Public\\Documents\\IDRISI Tutorial Data\\Using Idrisi\\'

This code shows two methods contained within the default project object. As you can see, getProjectDirs() method retrieves a list of all directories linked to the project, with the first directory as the working directory and any subsequent directories as resource directories. The getWorkingDir() method retrieves a string of the working directory file path. Knowing this, what do you think default.getResourceDirs() would return?

>>> default.getResourceDirs()

[]

If you guessed an empty list, then you guessed correctly, for there are no assigned resource directories within the project. However, we can assign new resource directories to the project object so long as we have a valid file path name. Let us add the \IDRISI Tutorial Data\Introductory GIS folder to the project as a resource folder.

>>> intro\_gis = ‘C:\Users\Public\Documents\IDRISI Tutorial Data\Introductory GIS’

>>> default.addResourceDir(intro\_gis)

Do you notice anything different when entering default.dirlist into the prompt? If you noticed that the file path for the input resource directory has been appended to the existing list, then you are correct! Now try the same very quickly for default.resdirs and try to guess what will happen. Now just for a moment open up Idrisi. In Idrisi Explorer under the projects tab, you can see that the directory path stored within the intro\_gis variable was added to the project and that you may now start accessing files from it.

We will be creating another project using the intro\_gis variable later, so let us remove the resource directory from the directory list. We can do that using the remResourceDir(path) method as shown below.

>>> default.remResouceDir(intro\_gis)

While not shown in this section, another useful method for managing your working and resource directories includes remAllResourceDir(path) to remove all resource directories within the project. If you would like to change the working folder of your project, you may use the setWorkingDir(path) method.

Now that you have an understanding of how projects operate within the Python shell using idrtools, the next section will demonstrate some useful tools to access files within project objects.

***Section 2-3: Organizing Files***

Now that we have gone over some useful methods for working with project objects and saving information concerning directories as variables, it is time to explore some additional methods that allow for organization of files within idrtools. If you are just beginning this tutorial, perform all tasks mentioned previously by import all items from idrtools, set up your project object using the project titled “default” using the code provided below.

>>> from idrtools import \*

>>> default = IdrisiExplorer()

Now that we are caught up, let’s look at some files. Create a variable called workFiles and assign it to the ListFile method found within the default project object. For now, we will only enter in the workdir variable as a parameter. We will be looking at other parameters a little later.

>>> workdir = default.getWorkingDir()

>>> workFiles = default.ListFile(workdir)

For the sake of document space and saving trees/memory/coal etc., this tutorial will not be displaying the results of this method, for directories with numerous files will take up much of the screen. As you have noticed, the output is in a list format. Take some time to look at the order of files in the list and compare it to the order of files displayed in Idrisi. Notice how they are all the same? Pretty cool huh?! If you do not like to view the files in a list, you may use the DisplayFileList method within the project object to neatly display your files and their associated index number in the python shell. The index number will allow you to easily access the file name string in the file list for future analysis purposes.

>>> default.DisplayFileList(workFiles)

If you wanted to view all of the files within this current project, you may use the ListAllFiles method within the project object to create an embedded list of all files within your project directories. To access each file, the first index signifies the directory, and the second index signifies the file name itself.

>>> default.ListAllFiles()

Notice the extension given for all of the files. Does this seem familiar? Return to Idrisi and click on the Filters tab in Idrisi Explorer. The file types listed in your workFiles list have the same extension as those allowed by the filter. To see which file types are being filtered, you may access the file extension filter by opening the filters variable. You may also add file type extensions using the addFileType function. For starters, let’s add the raster documentation file extension (‘rdc’) to the list.

>>> filters #Default parameters

[‘rst’, ‘rgf’, ‘tsf’, ‘vct’, ‘vgf’, ‘vlx’, ‘vgf’]

>>> addFileType(‘rdc’)

>>> filters #rdc extension added

[‘rdc’, ‘rst’, ‘rgf’, ‘tsf’, ‘vct’, ‘vgf’, ‘vlx’, ‘vgf’]

Now open up the filters variable and reassign your workFiles variable using the ListFile method with only the working directory as the parameter. Notice how the list has nearly doubled in size with the inclusion of the ‘rdc’ extension. However, it is important to keep in mind that if you store a list of files as a variable, you must manually update this variable every time that you add or remove a filter. Also, the filters are not linked to the Idrisi filters tab, so you will have to change those filters manually as well. To save some viewing space (and sanity while we are at it), use the removeFileType function to remove the ‘rdc’ extension from the list and reassign the workFiles variable once again.

>>> removeFileType(‘rdc’)

Now that we have a variable that houses all of the files within our working directory, it is now time to mess around with wildcards. Wildcards are commonly used in searches to find files with a specified sequence of characters. If you open up your workFiles list, you will notice that there are many files with the name ‘sierra’ in them. Let us grab only those files and create a list out of them. We can do this by putting ‘sierra’ into the wildcard parameter of the ListFile method. Create a variable called sierras and assign it a list of files using the ListFile method while inserting the ‘sierra’ string as the wildcard.

>>> sierras = default.ListFile(workdir, ‘sierra’)

Notice that only files with the ‘sierra’ string sequence present within them made the cut. Also notice how the files are all lower cased. This is because the third parameter of the ListFile method, which asks for case sensitivity, is set to True. Now create a new list variable called sierras2. Follow the same directions with the ListFile method above, but this time, change the case parameter to False.

sierras2 = default.ListFile(workdir, ‘sierra’, False)

Look at your output list for sierras2. Do you notice what new files were added to the list? This method allows for much flexibility in organizing files within your Idrisi projects and will be invaluable for your future work with this package and in this tutorial. The next section will provide a short lesson on reading and creating Idrisi-specific files, so stay tuned.

***Section 2-4: Reading and Writing Idrisi-specific File Types***

A crucial feature for organizing files and carrying out analysis using idrtools is the creation of numerous Idrisi-specific file types. While there are dozens of different extensions for Idrisi files, in this section, we will briefly go over reading and writing one of the most crucial file types used in Idrisi, raster group files. To begin, if you are just joining us, please instantiate the default project object with the current default Idrisi project. Once that is completed, identify the workdir variable as the working directory file path, and create the file list workFiles using the ListFile method within the project object.

>>> from idrtools import \*

>>> default = IdrisiExplorer()

>>> workdir = default.getWorkingDir()

>>> workFiles = default.ListFile(workdir)

Similar to the previous exercise, let us isolate only the files with the wildcard ‘sierra’. Let us call this list once again sierras. For this exercise, leave the case parameter as True.

>>> sierras = default.ListFile(workdir, ‘sierra’)

Now that we have all of our files with the ‘sierra’ wildcard, it is time to make a raster group file. To do this, we will call upon another default object within idrtools called IdrisiFiles. This object is an instantiation of the class IdrisiFiles that allows for convenient and consistent access of idrtools file writing methods without having to create a new object every time you start a new session. From the (IdrisiFiles) object, or file object, call the writeRgf method, and input the directory where the raster group file will be produced, workdir, the list of files to be made into a raster group file, sierras, and enter the name of the output raster group file, ‘SIERRA’. Notice how we did not have to insert a file extension in the output raster group file name. Pretty convenient huh?

>>>IdrisiFiles.writeRgf(workdir, sierras, ‘SIERRA’)

Once you have completed this step, open Idrisi and refresh the files tab in windows explorer. You should see that the SIERRA.rgf file has been added to the list. Please open it and open all of the images to make sure that the raster group file creation was successful. If you were to list all of the files in the Python shell using the ListFile method, you would see that the list now contains ‘SIERRA.rgf’. Please test that out now.

>>> default.ListFile(workdir)

Now that we have a raster group, let us see if we can read the files currently inside. To do this, we will be calling upon the (readRgf) method within the files object. This method only requires the name of the raster group file (no rgf extension necessary).

>>> IdrisiFiles.readRgf(‘SIERRA’)

Notice that the output of this method is identical to the code written below:

>>> default.ListFile(workdir, ‘sierra’)

However, one is reading all files within a raster group file, while the other is identifying all files within a directory with a given wildcard. The (writeRgf) and (readRgf) methods are particularly useful for analysis if you would like to organize particular files for later analysis of for executing or building tools that require a raster group input file. Other file types that can be produced with idrtools that will be discussed later on within this tutorial include reclass files (rcl) and attribute values files (avl).

***Section 2-5: Reading and Updating File Metadata***

This section is currently under development…please stand by…

***Chapter 3: Using Modules for Data Display, Analysis and Batch Processing***

*Sections*

Displaying Files using idrtools

Using Modules for Batch Processing

Using Modules for Analysis and Preliminary Tool Building

The data for this section may be found in a directory ending with the file path ‘\IDRISI Tutorial\Using IDRISI’ and \IDRISI Tutorial\Introductory GIS If the installation of the IDRISI program was set to default, these directory will be on the same drive as the installed IDRISI program.

***Section 3-1: Data Visualization and Displaying Files Using idrtools***

Now is the moment that you have been waiting for. Having achieved a fundamental understanding of the idrtools UI, it is now time to go over the use of modules, a crucial process for using the Idrisi software. Traditionally, modules in Idrisi could only be run using the RunModule method within the Idrisi API. First let’s demonstrate this through an example. For those just joining this tutorial exercise, please enter the following lines to catch up.

>>> from idrtools import \*

>>> default = IdrisiExplorer(‘default’)

>>> workdir = default.getWorkingDir()

>>> workFiles = default.ListFile(workdir)

>>> sierras = default.ListFile(workdir, ‘sierra’)

Now, let’s produce a true color composite with the composite module. Using the traditional means for accessing the RunModule method within the Idrisi API, the code would look something like this.

>>> import win32com.client

>>> api = win32com.client.Dispatch(‘idrisi32.IdrisiAPIServer’)

>>> api.RunModule(“COMPOSITE”, ‘sierra1\*sierra2\*sierra3\*

TCC\_test1\*1\*2\*1\*3’, 1, ‘’, ‘’, ‘’,‘’, 1)

>>> api.DisplayFile(‘TCC\_test1’)

As you may notice, the second parameter within the RunModule method is the Idrisi Macro Language script for running the function. This is how all Idrisi modules are called using the RunModule method. The idrtools package greatly simplifies this process by encapsulating this script within a function containing fewer parameters. The use of the composite module in idrtools is shown below:

>>> modules.composite(sierra[0], sierra[1], sierra[2], ‘TCC\_test2’)

This script performs the identical operation as above, but with much less of the headache. To display the file, use the displayLauncher method within the modules class. This method requires the name of the file to be viewed and the file type, either raster or vector. While the final parameter is the color palette, no palette is necessary for files with a RGB24 data type.

>>> displayLauncher(‘TCC\_test2’, ‘rst’)

Now that we have visualized the true color composite image, let’s now try to display an image that does not have a data type of RGB24. Let us display the digital elevation model of the sierra image.

>>> displayLauncher(‘sierradem’, ‘rst’)

Now switch back into Idrisi. Notice how the image is displayed using the “quant” palette. Now open up the palette variable in Python.

>>> default\_palette[0]

‘quant’

Notice how the default palette set is ‘quant’. To change this, simply type in the string name of the palette you wish to automatically display your files in. Unfortunately, unlike the default project mentioned in chapter 2, the default palette does not keep when changed. You will have to change it manually in the script and save it if you want to keep the change permanent. For now, change the palette variable in the shell to ‘greyscale’ and display the file again.

>>> displayLauncher(‘sierradem’, ‘rst’, ‘greyscale’)

Now that we have both files displayed, let’s re-display it using the ortho module in Idrisi. To do type in the following below. We will be using the ‘sierradem’ image as the surface, the ‘TCC\_test2’ image as a drape, the output file will be named ‘TCC\_ortho’, and we will be using a 800 x 600 resolution with the third parameter 2.

>>> modules.ortho(‘sierradem’, ‘TCC\_ortho’, 2, TCC\_test2’)

>>> displayLauncher(‘TCC\_ortho’, ‘rst’)

Pretty cool huh?! Unfortunately Idrisi is not widely used for cartographic purposes, so while you may not frequently use these modules for map presentation, this exercise is meant to familiarize you with the module layout in a very basic sense.

***Section 3-2: Using Idrisi Modules for Batch Processing***

Now that you have explored some basic visualization tools with this package, it is now time to explore some of the powerful tools and options available to users through the use of modules. We will begin by discussing one of the most important, but potentially redundant processes in GIS Analysis, batch processing. For this tutorial, we will be using the January Madagascar Files from the Introductory GIS directory. If you are just joining us, please follow the code below. If you are continuing from the previous exercise, skip this code and proceed to the next step.

>>> from idrtools import \*

First things first, we will create a new project for this exercise. Please follow the code below to do so.

>>> intro\_gis = ‘C:\Users\Public\Documents\IDRISI Tutorial Data\Introductory GIS’

>>> openNewProject(intro\_gis)

>>> introgis = IdrisiExplorer(‘Introductory GIS’)

Cool, you have a new project. Remember to return to Idrisi and be sure that the radio button is assigned to this project before you start working. Now that the project is assigned, it is time to look at the files. However, we only want the Madagascar NDVI files, so we will incorporate a wildcard ‘mad’ in our search. To make things more interesting, we will create a Raster Group File containing all of the NDVI imagery right off the bat. Call the Raster Group File ‘MADNDVI.rgf’.

>>> workdir = introgis.getWorkingDir()

>>> IdrisiFiles.WriteRgf(workdir, introgis.ListFile(workdir, ‘mad’), ‘MADNDVI’)

From here, we can start building some handy dandy tools for batch processing. First, let’s create a function to carry out all of our initial data conversion. This function will allow us to convert the Madagascar NDVI files from byte into ratio values using the formula:

NDVI = (Dn \* 0.0028) – 0.05

Where Dn is the initial scaled digital number in byte format. First we must define the function with the input raster group file and our output prefix string.

>>> def byteToRatio(input\_rgf, output\_prefix):

Now we must perform the mathematical operations using the Idrisi modules. However, we must first read the Raster Group File and perform this transformation on all files. Before we move on after this, we should also make an empty list for our future output files from this process. This will allow us to use this list to create an additional list for our output files such that we may create another Raster Group File.

files = IdrisiFiles.ReadRgf(input\_rgf)

output\_files = []

Now that we have our files in list form and an output list, it is time to start working. First we will need to run the files through a loop, then we will need to perform a scalar multiplication operation of 0.0028 on the file. Then we will need to subtract 0.05 from each image using the scalar module yet again. This means that we will have a lot of extraneous intermediary files to fill our working folder and bother us, right? WRONG!!! We can create a dump directory for which such files can be added and removed easily. Let us start with some code that will allow us to create a temporary ‘dump’ directory to store our extra files.

dump = addDumpDir(workdir)

Now that we have the dump directory, let us perform the mathematical equation expressed above for each file. In between each scalar run, we shall name our output file as the concatenation of our output prefix defined as a parameter and each file name. Once the last scalar run has been performed, we shall append the name of the file to our output\_files list

for image in files:

modules.scalar(image, dump+’tmp001’, 3, 0.0028)

output = output\_prefix + image

modules.scalar(dump+’tmp001’, output, 2, 0.05)

output\_files.append(output)

Great! This function is really starting to come together! Just in case you may not know, the inputs for the scalar module are the exact same as the Idrisi Macro Language inputs. The first input to the function is the input file, the second input is the output file, the third input is the operation, and the fourth input is the value of the operation. The 3 for the operation parameter indicates multiplication for the first scalar run, and the 2 indicates subtraction for the second scalar run. Now we shall think about our end game and bring forth our function outputs. The output will be the name of the Raster Group File that we will create with our last line of code.

IdrisiFiles.WriteRgf(workdir, output\_files, output\_prefix +input\_rgf)

Now, like we do, or should do at least once a week, it is time to throw out the dump directory using the removeDumpDir() function. To finish off this function, we will return the name of the output Raster Group File.

removeDumpDir(dump)

return output\_prefix+input\_rgf

In total, the function byteToRatio() should look a little something like this.

>>> def byteToRatio(input\_rgf, output\_prefix):

files = IdrisiFiles.ReadRgf(input\_rgf)

output\_files = []

dump = addDumpDir(workdir)

for image in files:

modules.scalar(image, dump+’tmp001’, 3, 0.0028)

output = output\_prefix + image

modules.scalar(dump+’tmp001’, output, 2, 0.05)

output\_files.append(output)

IdrisiFiles.WriteRgf(workdir, output\_files, output\_prefix +input\_rgf)

removeDumpDir(dump)

return output\_prefix+input\_rgf

Now let us run it using the data that we do have. We will use the name of the ‘MADNDVI’ Raster Group File that we just created and the string prefix ‘Rat\_’ indicating a conversion of byte to ratio values.

>>> madndvi\_rat = byteToRatio(‘MADNDVI’, ‘Rat\_’)

This should assign a Raster Group File titled ‘Rat\_MADNDVI’ in which all files contain ratio NDVI values to the variable madndvi\_rat.

Now, let us say that we want to find the mean value of NDVI. How should we go about doing that? We could do that through some additional batch processing with our new NDVI dataset. Let us first define a function to calculate the mean NDVI for all Madagascar images. The two parameters for this function will be the input Raster Group File and the output file name.

>>> def meanRaster(input\_rgf, output\_img):

Now before we can begin calculation, we need to read all of our files in the Raster Group File. We can then use the spatial dimensions of our first file to create a blank image using the initial module. We should also make another dump directory such that we can easily discard any extraneous files.

files = IdrisiFiles.ReadRgf(input\_rgf)

dump = addDumpDir(workdir)

blank = dump+’blank\_ndvi’

modules.initial(blank, 2, 1, 0, files[0])

Now that we have a blank image upon which we may sum our NDVI results, we can now find the sum of the NDVI files using a for loop and the overlay module.

for image in files:

index = files.index(image)

tmp\_output = dump +‘SUM\_’+str(index+1)

modules.overlay(blank, image, tmp\_output, 1)

blank = tmp\_output

This for loop will start with a blank image with the same spatial extent of our Madagascar imagery. From there it will add the blank image to the Madagascar image and create a temporyr output tmp\_output. Once the overlay is complete, the tmp\_output file name is assigned to the blank variable, and the process repeats until there are no images left. From there, it is time to find the average of all the files using the scalar module. Once that is completed, we dump the extra files and return the name of the output image and display our image.

modules.scalar(blank, output\_img, 4, len(files))

removeDumpDir(dump)

displayLauncher(output\_img, ‘rst’)

return output\_img

To run the function, we will enter our madndvi\_rat variable as the input Raster Group File and name the output as ‘Mean\_MadJanNDVI’. Call this function using the variable madndvi\_mean.

>>>madndvi\_mean(madndvi\_rat, ‘Mean\_MadJanNDVI’)

Let us recap for a second and print out the entire script as it stands.

>>> def meanRaster(input\_rgf, output\_img):

files = IdrisiFiles.ReadRgf(input\_rgf)

dump = addDumpDir(workdir)

blank = dump+’blank\_ndvi’

modules.initial(blank, 2, 1, 0, files[0])

for image in files:

index = files.index(image)

tmp\_output = dump +‘SUM\_’+str(index+1)

modules.overlay(blank, image, tmp\_output, 1)

blank = tmp\_output

modules.scalar(blank, output\_img, 4, len(files))

removeDumpDir(dump)

displayLauncher(output\_img, ‘rst’)

return output\_img

Pretty nifty huh?! Well that does it for our lesson on batch processing using idrtools. In the next chapter, we will learn how to use idrtools to produce more analytical tools for problem solving purposes.

***Challenge:***

Using the meanRaster function as a base line, create two functions, varianceRaster and stdevRaster to calculate the per-pixel variance and standard deviation across each Madagascar image in the time series.

***Section 3-3: Using Modules for Analysis and Preliminary Tool Building***

Coming soon to an idrtools tutorial near you…

***Chapter 4: Idrisi Tool Development with “mytools”***

*Sections*

Using Tools in mytools

Creating a Receiver Operating Characteristic Curve

Helpful built-in Python modules and 3rd Party Packages

Chapter currently under development…