CORNELL SATELLITE REMOTE SENSING
SUMMER 2023

BRUCE MONGER (INSTRUCTOR)
BCM3@CORNELL.EDU

DANIELLE MANGINI (TEACHING ASSISTANT)
DMM433@CORNELL.EDU

NOUR KASTOUN (TEACHING ASSISTANT)
CNK32@CORNELL.EDU

PHIL BRESNAHAN (GUEST INSTRUCTOR)
BRESNAHANP@UNCW.EDU

SARA RIVERO CALLE (GUEST INSTRUCTOR)
RIVERO@UGA.EDU

TABLE OF CONTENTS

PYTHON PROGRAMMING METHODS

RUNNING PYTHON: INTERACTIVE PYTHON OR THE PYTHON INTERPRETER

1. ***You can run python in <u>Interactive Mode</u>*** at the command line after you launch python in a *Unix Terminal Window*

   Running python interactively is a really great way to quickly explore how specific python functions work…

2. ***You can also run python in Text File <u>Interpreter Mode</u>*** to execute text documents containing a series of saved python commands

   Running python in the interpreter mode on a text file of python commands is really great for running elaborate python programs and/or running the same commands over and over from day to day…

WHERE TO SAVE THE PYTHON PROGRAMS THAT YOU WRITE

3. The Unix environment, under which python will be running, needs to be given explicit instructions describing where the python programs you create will be located (*i.e., what file directory path you saved your python programs in*).
4. The file directory information is given in special Unix startup *text file* named ***.bashrc*** (*the dot at the beginning of the name is part of the file name*) that is located in your home directory (**~/**).
5. The ***.bashrc*** file sets ups a bunch of *<u>system variables</u>* (similar to aliases) that, among other things, tells python the file directory path being used to store the python programs you create. **It runs each time you open a new terminal window**.
6. I have already specified the path to your python programs to be: ***~/python_programs***
7. *Any new programs you make in this class must reside within this directory or python will not be able to run them. If you want to check the path… Open a Terminal window and type:* ***echo $PYTHONPATH*** *-- and then <return>*
8. Note: Macs run a file called ***.bash_profile***, but I have added a command in this file to run the ***.bashrc*** file.

--- Very Important Notice ---

Background: The upcoming python tutorial references a file directory path referenced *relative to* ***<u>your home directory</u>***. Depending on how the computer system was initially configured for this class, it is likely that each person will have a *unique home directory* that is named with their unique login username.

The Unix *<u>alias</u>* that represents everyone's home directory is given by: **~/** So, *~/data/tutorial_exercises/* and ***/Users/your_home_directory/****data/tutorial_exercies/* are equivalent.

In the upcoming python tutorial exercises, the data files that are accessed within your python programs are described using the ***~/ alias***. **However**, since python ***does not*** recognize ~/ as equivalent to your home directory, **the following must be done**:

Whenever you come across ~/ as part of the instructions in this tutorial, **you need to** replace it with your actual explicitly written home directory.

NOTE, the explicit name of your home directory can be found by opening a unix terminal window and *typing* ***pwd*** at the unix prompt.

IMPORTANT: for example, if the instruction shows the following:

fname= '~/data/aqua/A20181231423.CHL.GLOBAL.nc'

You should replace it with the following:

fname= ***'/your_explicit_home_directory/***data/aqua/A20181231423.CHL.GLOBAL.nc'

WRITING YOUR FIRST PYTHON PROGRAM IN TEXT FILE INTERPRETER MODE

1. Launch the *Text Editor* Application

2. Following in a long-long-long line of programming tradition, your first piece of code will be to simply print *"Hello World*" to the Unix Terminal window. To do this, type the following line of code into the open text document.

   ***Print('*Hello World')**

3. Okay you now need to *Save* the content of your text file by clicking the save button at the top of the open text editor window and then do the following:

   a. Navigate to your *home directory*
   b. Then navigate into the *python_programs* directory
   c. Then **make a new directory** under the python_programs directory called: ***tutorial_exercises***
   d. And then save this first program with the name: ***myprogram1.py***

4. The program you have created is run from the *Unix Terminal Window*. So, launch the *Terminal Application* and ***c***hange ***d***irectories to where your new python program resides by typing the following in the open terminal window:

   cd ~/python_programs/tutorial_exercises <*return*>

5. Now type: ***ls*** to list the files in this directory. You should see the file *myprogram1.py* listed.

6. Now to run your new program, type the following into the terminal window:

   python myprogram1.py

7. You should see the words: Hello World printed to the terminal window.

8. Congratulations on running your first python program!!

WRITING YOUR FIRST PYTHON COMMAND LINE IN INTERACTIVE MODE

1. Launch Terminal
2. To launch python in interactive mode, type: **python <RETURN>**
3. Type: **print('Hello world')** then press the **<RETURN KEY>**
4. You should see the words: Hello World printed to the terminal window.
5. Exit python by typing **exit()** then press the **<RETURN KEY>**

Now that you have seen how easy it is to work with python in the simple example above, we are going to take a pretty big diversion next and dive into some important/lengthy discussion of some python background information and then we will be set to work with more elaborate python examples…

PYTHON MODULES AND PACKAGES

The simple program you created above used python's built-in ***print*** function. There are a finite number of built-in python functions and procedures. **However**, other people/organizations have written an almost unlimited number of additional python functions and procedures that you can add to your own python program.

These additional functions and procedures are organized into ***modules*** and ***packages*** and then *explicitly* imported into your python program with the ***import*** command.

There are several ways to import modules and packages into your program. The follow examples assume a hypothetical module called ***module***

1) You can load the **entire module** with all of the contained functions. The following are methods of importing the entire module. *Generally speaking the first option below is the best choice…*

import module as name
Example: *import numpy as np*
Calling a Function: *np.array(…)*

import module
Example: *import numpy*
Calling a Function: *numpy.array(…)*

from module import *
Example: *from numpy import **
Calling a Function: *array(…)*

2) You can also load just a **single function** from the module

from module import function
Example: *from numpy import array*
Calling a Function: *numpy.array(...)*

3) Packages are an assemblage of modules and you can import the **whole package** or a **single module from the package**

import package.module as name
Example: *import os.sys as sys*
Calling a Function: *sys(...)*

Some Commonly Used Modules:

numpy and **scipy** – provide most of the common methods from Matlab, as well as important types like arrays. (non-default, included in installation instructions)

matplotlib – provides methods for creating figures, with almost identical syntax as Matlab (non-default, included in installation instructions)

os – provides operating system commands, like changing directories and moving files around

sys – provides methods for accessing system information, passing arguments to functions, and controlling input/output/error streams

io – provides input/output methods for reading and writing files

subprocess – provides methods for calling Unix tools and programs from the terminal, as well as reading their output streams

NOTE: It can often be helpful to see the spelling of all the functions contained in a given module. This can be done with the ***dir*** command. See the example below for using the interactive python approach.

i) In the open terminal window type: ***python*** to launch python in *interactive mode*.
ii) Now import the math module by typing: ***import math***
iii) Now type: ***dir(math)*** to get a listing of all the functions contained in the math module.
iv) Exit the interactive python mode by typing: ***exit()***

STRINGS, TUPLES, LISTS AND ARRAYS

I am pretty new to python and I have to say that the concept of *Tuples* and *Lists* (*versus Numerical Arrays- i.e., vectors and 2D matrices of numbers*) gave me a fair bit of confusion in getting started with python. So, I ***strongly encourage you*** to pay attention here and really get these three "*Structures*" down clearly in your mind.

Strings -- should be pretty clear to most of you who have programmed before. They are just one long running sequence [*string*] of ASCII [*keyboard*] characters **denoted in a python program with single or double quotes**. For example: 'Hello World' and "Hello World" are equivalent *strings*. More will be said later about manipulating strings, but for now I will just say two things about Strings:

- You <u>can</u> index the position of each character in the string similar to a numerical vector
- You <u>cannot</u> change the letters within the string (***<u>Strings Are Immutable</u>***).

Tuples -- are a sequence of almost anything you can imagine (e.g., ascii characters, integer numbers, floating points numbers and even a sequence of individual vectors and/or arrays or ***<u>a mix of all these data types</u>*** in the same sequence). Tuples are **denoted in a python program with *parentheses***. For example: (1, 2, 3.7, 'hello world', [1, 3, 27], 4.0, 15) is a *Tuple*.

An important point to note is that ***<u>Tuples Are Immutable</u>*** – Once the Tuple sequence is created it (the sequence) cannot be altered (but you can append two Tuples together if you want to create a new longer Tuple).

- I like to think of Tuples as fancy *Strings* since both Strings and Tuples are **Immutable Structures**.

Lists -- are getting closer to what you might already know about *Numerical Arrays**, but Lists are definitely <u>NOT</u> numerical arrays***. Lists are a running sequence of almost anything you can imagine (e.g., ascii characters, integer numbers, floating point numbers and even a sequence of individual vectors - and/or arrays or ***a mix of all these data types*** in the same sequence). Lists are **denoted in a python program with square brackets**. For example: [1, 2, 3.7, 'hello world', [1, 3, 27], 4.0, 15] is a *List*

An important point to note is that ***<u>Lists Are Mutable</u>*** – Individual objects in the list can be changed (or removed). *Lists* have a lot of advantages over numerical arrays (see next) when it comes to certain *List* manipulations.

- **But…** for most hardcore math stuff on *1D vectors* or large *2D matrices* (i.e., satellite images) **you will almost always want to use *numerical arrays* and <u>not</u> *Lists***. See the difference between numerical arrays and Lists here: http://bit.ly/1KlkPvd

Numerical Arrays -- are created using the added ***numpy* module**. *Arrays must have a consistent data type* (e.g. all integers or else all floating point numbers or else all strings).

- Numerical Arrays are held in computer memory as *C* language arrays in very compact form, so they are <u>*data memory efficient*</u>.

- Math operations on *Arrays* can be <u>*vectorized*</u> so they *can have <u>very fast computational speeds</u>*. Consequently, whenever you are doing serious math you should use numpy numerical arrays.

Note -----> It can be helpful to know what kind of data *structure* you are working with in a python program you have written. Built-in or 3rd-Party python programs can often return data *structures* that are *Tuples* or *Lists* instead of the more familiar *numerical array* that you might have expected, and you might then end up misusing the returned data structure (*e.g., trying to change the value of an element in a returned Tuple*) and force an error in your program…

To determine the type of data structure you are working with, use the following command in your python program or at the python prompt:

Write: ***type(the name of the structure you want info about)***

SOME VERY BASIC MATH POINTS

In this section we will examine some simple math operations and also see how python works when run in ***Interactions Mode*** Later we will run more complex math operations written in text files and run with python in ***Interpreter Mode***.

To run Python in Interactive Mode:

1. Open a new terminal window
2. Then type: ***python*** --- to start python
3. You should see: >>> which signifies that python is running *Interactively*.
4. Now type the following and see what you get:

```
>>> 2*3
>>> 2**3
>>> 5+2*5
```

Boolean Operations and Comparisons

Boolean Expressions are expressions that evaluate to True or False. Here are some common operations you need to know:

x or y	(same as x | y)
x and y	(same as x & y)
not x	

Here are some common Boolean comparisons:

```
<       # less than
<=      # less than or equal to
>       # greater than
>=      # greater than or equal to
==      # equal to  (Note: this is 2 equal signs)
!=      # not equal to
in      # within
```

Of course, all of these operations can be strung together into more complex logical expressions that will be useful when you get to *IF* statements and/or *While* loops. You can test some of these Boolean expressions with the following statements:

```
>>> 2==2.0
>>> 2<=5
```

```
>>> (5.0/2.0)!=2.5
>>> (2==2.0) & (5>3)
>>> (2.0>3.0)|(6.0==6.0)
```

SOME BASIC STRING MANIPULATIONS

One of the many benefits of Python is that it provides you with dozens of useful and simple *string methods*. Strings can be denoted by either *single quotes* (‘ ’) or *double quotes* (“ ”).

NOTE: Having a strong working understanding of string manipulation is ***absolutely vital*** to writing data processing scripts – **So pay good attention here**!

```
>>> print(‘Hello’)
>>> print(‘World’)
>>> new_string = ‘Hello’ + ‘ ‘ + ‘World’
>>> print(new_string)
```

Each individual character in a *String* sequence (and the same goes for *Tuples, Lists* and *Arrays*) can be identified by its **index position** starting with position zero to the far left. So, for example the string ‘hello world’ has an *h* at index position 0 and the letter *w* at index position 6.

The following notation allows you to extract a substring from within a larger string (a process referred to as ***Slicing***):

```
>>> substr = fullstr[start_index:end_index]
```

When you ***Slice***, the substring will include everything **starting** at index *start_index* and up to, **but *not* including,** *end_index*. <---- **SLICING / SUBSCENING IS A BIG DEAL!**

Try out the following string slicing commands to get a feel for indexing:

```
>>> s = 'Hello World.txt'              # creates a new string
>>> s[1]
>>> s[1:5]
>>> s[:5]
>>> s[5:]
>>> s[-3:]
```

Notice that if you do not include a start_index (or end_index), python will automatically start (stop) at the beginning (end) of the string. You can also use negative numbers, which wrap around and count backwards from the end of the string. This is particularly useful when you want to get the file extension (.txt, .png, etc.) from a filename. Another **very useful operator** is the ***in*** operator (more will be said about the ***in*** operator later). Try the following:

```
>>> filename = 'S1998148172338.chlor_a.f999x999'
>>> '1998' in filename
>>> 'c' in filename
>>> 'w' in filename
```

In addition to concatenation and slicing, there are dozens of other useful string methods. A full list is here: https://docs.python.org/3/library/string.html

Here are a few of the most useful string methods, try to guess how they work before looking up their specifications:

```
>>> s = '  crazy sample\n'          # \n is the symbols for a newline (i.e., < return> key)
>>> print(s)                        # notice the newline…
>>> s = '\ncrazy sample  '          # puts the newline (\n) before printing the string
>>> print(s)
>>> print(s.strip())                #removes white space before and/or after text
>>> print(s.split('a'))             #splits the string whenever there is an 'a'
```

I highly recommend checking out the link above for the string module library, as you will certainly use many of these in the future.

SPLITTING THE FULL FILENAME INTO THE *DIRECTORY PATH* AND *BASENAME*

There is one more function that is closely related to the *string functions* given above. Very often you will read in a satellite filename and use some of the information contained in the filename itself. For example, the full file name:

~/data /tutorial_data/SEASTAR_SEAWIFS_MLAC.20030921T171049.L2.OC.nc'

The file name identifies it is *S*eaWiFS data that was collected in *2003* on September 21 at precise time: **T***hhmmss* UTC.

An effective way to read in the satellite data type and the Year/Date is to **first break off the path part of the file name** and then use the string functions above to pick off the **SEAWIFS** and the *2003* and the *0921 (i.e., Sept 21)*.

To break the path from the basemane part of the full file name you will use python's ***os*** function that needs to be loaded into python before being available to use. Use the following as an example:

Open a Unix Terminal window and launch python and then use the following commands.

```
>>> import os
>>> filename = '~/data/tutorial_data/SEASTAR_SEAWIFS_MLAC.20030921T171049.L2.OC.nc'
>>> dir_name =      os.path.dirname(filename)
>>> base_name =   os.path.basename(filename)
>>> print(dir_name)
>>> print(base_name))
>>> print(base_name [8:15])
>>> print(base_name [21:25])
>>> print(base_name [25:29])
```

To Exit Out of Python Interactive Mode Type: ***exit()*** *---and then <return>*

SIMPLE EXAMPLES OF WORKING WITH TUPLES, LISTS AND ARRAYS

Tuples -- You will only occasionally come across *Tuples* in this class so I will just remind you that *Tuples* are *immutable* and are **created using parentheses.** I have never run into any cases of purposely making *Tuples* in my python programs, ***but*** there are certainly occasions when some built in functions return a *Tuple,* so you definitely need to know they exist!

- A classic example is given later when the ***numpy where function*** is discussed. *The where function acting on a 2D array (aka. satellite image) returns a* ***2-element Tuple: (a row vector, a column vector****) of good locations****.***

Lists – *Lists* can be nice for indexing and working with **sequences of different data types** (*i.e., floating point and integer and ascii strings in the same List*), **but** you are more likely to be working with ***numerical arrays*** in this class. However, *Lists* are often returned from other functions in python, so you definitely do need to know a bit about how *Lists* work.

A *List* is created with square brackets *[ ].* *Lists* can be *Indexed* and *Sliced* in the same way as *Strings* are sliced. They also support some operators, which you can experiment with below:

1. Open a new *Terminal Window* and **type: *python*** to start python in *Interactive Mode* and then type the following...

```
>>> x = [8, 'cat', 5.0]
>>> print(x)
>>> x[1:]
>>> x + 2       # error
>>> x + [2]     # append another list
>>> x*3         # may not do what you expect
```

2. Just like strings, *Lists* have many of their own methods. Here are a few common ones, which can be found at https://docs.python.org/3/tutorial/datastructures.html

```
>>> x = [8, 'cat', 5.0]
>>> x.append(2)
>>> print(x)
>>> x.insert(1, 'dog')
>>> print(x)
>>> x.remove(8)
>>> print(x)

>>> y = [4, 2, 7, 2, 9, 2]
>>> y.index(2)
>>> y.count(2)
>>> y.sort()
>>> print(y)
>>> y.reverse()
>>> print y

>>> z = range(2, 7)
>>> print(z)
```

Numerical Arrays – Basic python does not have numerical array data types, **but** the numerical array data type does become available when you import the ***numpy*** **(Numerical Python) module**.

IN ORDER TO USE NUMPY ARRAYS, YOU MUST FIRST IMPORT THE NUMPY MODULE

```
>>> import numpy as np
```

Convert a *List* to an *Array*

```
>>> A = np.array([1, 2, 3, 4, 5])
>>> print(A)
```

Convert a *Tuple* to an *Array*

```
>>> B = np.array((1, 2, 3, 4, 5))
>>> print(B)
```

Convert a 3x3 List to a 3x3 Array

```
>>> C = np.array([[1.2,2.1,3.0], [4.8,5.0,6.0], [7.0,8.2,9.0]])
>>> print(C)
```

Create a 3x3 array filled with zeros
Note the use a Tuple to prescribes the desired dimensions

```
>>> D = np.zeros((3,3))
>>> print(D)
```

Create a 10x10x10 array filled with ones

```
>>> E = np.ones((10,10,10))
>>> print(E)
```

Create 5x5 array filled with sequence 0 to 25

```
>>> G = np.arange(25).reshape(5,5)
>>> print(G)
```

Create 6x2 array filled with a sequence of 12 elements running from 1.2 to 4.6

```
>>> H = np.linspace(1.2, 4.6, 12).reshape(6, 2)
>>> print(H)
```

SOME SIMPLE EXAMPLES OF WORKING WITH NUMPY ARRAY

Mathematical addition, subtraction, multiplication and division are done element by element with numpy arrays. Note that numpy arrays are *indexed* and *sliced* the same way that was done with *Tuples* and Lists.

1. Launch python in Interactive Mode from the Terminal Window by typing: ***python***

```
>>> import numpy as np
```

```
>>> array1 = np.array([[1.2,2.1,3.0], [4.8,5.0,6.0], [7.0,8.2,9.0]])
>>> print(array1)
```

```
>>> array2 = array1 + 1.0
>>> print(array2)

>>> array3= array1 * 10.0
>>> print(array3)

>>> array4 = array1 + array3
>>> print(array4)

>>> array5 = array1*array3
>>> print(array5)
```

Keep in Mind that 2D numpy arrays (matrices) are indexed by [***row, col***] which is equivalent to [y, x] axis ordering instead of an [x, y] ordering that you might have encountered with other programming languages…

THE *NUMPY WHERE* STATEMENT <------ THE *WHERE* FUNCTION IS A BIG DEAL!

The ***where*** statement locates the indices (array element positions) within an array where the values at those positions satisfies user specified conditions. The following are good examples.

```
>>> import numpy as np
>>> a = np.arange(10)*10.0
>>> good_locations= np.where( (a > 20) & (a <=70))
>>> print(good_locations)
>>> print(a[good_locations])

>>> bad_locations= np.where( (a < 0) & (a > 100))
>>> print(len(bad_locations))
```

Now try this with a **2D matrix** of numbers

```
>>> a = np.arange(9)*10.0
>>> a=a.reshape(3,3)
>>> good_locations= np.where( (a > 20) & (a <=70))
>>> print(good_locations)
>>> print(a[good_locations])

>>> bad_locations= np.where( (a > 0) & (a < 100))
>>> print(bad_locations)
>>> print(len(bad_locations))
```

NOTE: For the case of a 2D matrix the ***where*** **statement** returns a ***Tuple*** **containing 2 vectors**. The two vectors are the ***row*** and the ***column pairs*** for the locations (indices) where the user specified conditions are True. Note that len(bad_locations) is 2 since it returns *a Tuple of two elements* for each dimension – but the Tuple elements might be two vectors of thousands of row, column pairs where the where conditions are True. To find out if no data was found you need to check one of the elements of the Tuple to see if its length is zero. **<--- THIS IS A BIG DEAL!!!!!!**

NOTE ---> The ***where*** statement gets used a lot, so **you should definitely pay attention** to how this function works!! **<--- THIS IS A BIG DEAL!**

There are many other array methods that can be found in the *numpy* documentation. Both the numpy and scipy (Scientific Python) modules are extra modules that need to be added to the default version of Python, which include countless methods for creating and manipulating arrays, computing linear algebra operations, integration, interpolation, and statistics. You can find the documentation for these packages at:

http://docs.scipy.org/doc/numpy/reference/
http://docs.scipy.org/doc/scipy/reference/

READING/WRITING DATA FILES

The basic design of most programs is to *read* in data from the hard drive, do some math on the data and *write* the new data results to the hard drive. So, reading and writing data files is a big deal. There are 3 basic file types we will be reading and writing in this class.

- Numerical Binary Files
- HDF/NetCDF Files
- Text Files

READING NUMERICAL BINARY FILES INTO PYTHON

1. Open a new text file and save as ***read_binary_program1.py*** in your ***tutorial_exercises*** directory and add the following code...

Note 1: The functions **fromfile** and **tofile** are part of the **numpy module** so you *must* load numpy to use this form of read and write.

Note 2: The functions used to display the 2D image to the monitor are contained in the **matplotlib.pyplot module** so it too *must* be imported.

Note 3: To read in a numerical binary file you must know *a priori* what kind of numerical values are being read (i.e., byte, integer, floating point) and the shape of the array being read (e.g, *the x,y dimensions of a 2D array of numbers must be known a priori*). The most common will be byte (int8, int16, uint32, float32 and float64). All the data types recognized by python are found here: http://docs.scipy.org/doc/numpy/user/basics.types.html.

```
----------------------------------------------------------------
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors

fname = '~/data/tutorial_data/S1998148172338_chlor_a.f999x999'
```

```
# open file and read data into a numpy array
# Line 1 opens the file - assumes data were previously written out to the hard drive as
# 32bit floating point numbers. Line 2 reads in all the data from the file into a numpy
# array. Line 3 closes the file.
# --
f = open(fname)
chl_array = np.fromfile(f, dtype=np.float32)
f.close()


# reshape data into 2D matrix to its original dimention
# All data are stored on the hard drive as one long series of numbers. After reading in the
# data you will need to reshape the long series of number (1-D) to the original 2-D
# dimensions.  NOTE: You would have to know a priori that the data had previously been
# written out as a 999x999 array
# --

print(chl_array.shape)

chl_array = chl_array.reshape([999, 999])

print('shape of data1 array: ',  chl_array.shape)


#The next lines of added code will image your 2D data array to the monitor…
```

IMAGING 2D DATA TO THE MONITOR

NOTE: For the case below you needed know a priori that the data was chlorophyll **and that chlorophyll is log normally distributed** (you also needed to know that the chlorophyll range is typically about 0.01 to 20 mg m^3 for your region of interest (Gulf of Maine). In almost all other cases besides chlorophyll (e.g., sst, wind, altimetry), the data **will be linearly distributed**.

For the case of **log normally distributed data**, you will use this format for plt.imshow.
plt.imshow(your_data, cmap=mycmap, ***norm=matplotlib.colors.LogNorm(##, ##)***)

For the case of **linearly distributed data**, you will use this format for plt.imshow.
plt.imshow(your_data, cmap=mycmap, ***vmin= ##, vmax= ##***)

```
# define color scale…
mycmap = plt.get_cmap('nipy_spectral').copy()  # load color rainbow palette
mycmap.set_bad('k')                            # set NaN values to display as black

# create and display the figure to your monitor with data in log scale…
plt.figure(1)
plt.imshow(chl_array, cmap=mycmap, norm=matplotlib.colors.LogNorm(0.01, 20.0))
plt.colorbar()
plt.show()
```

```
# Note: that the use of log10 is needed here only because chlorophyll concentration is Log
# Normally Distributed in space. For almost ALL OTHER data products (e.g., sea surface
# temperature, wind speeds etc…) you would NOT use the Log10 scaling….
```

2. Now save your python code text file and then go to the terminal window to run the program by typing:
   python read_binary_program1.py

   NOTE: you will either need to first go (***cd*** command) into the directory where your ***python read_binary_program1.py*** is saved, or you will need to provide the entire path of the program file, so python can find your script…

   The easiest thing to do is to ***cd*** to the directory where the python program was saved.

Writing Out Numerical Binary Data to the Hard Disc.

NOTE: *If* you *had* done some math on your 2D array above and created a new 2D array of values as a result (let's call the new data: *new_data_array*) and you wanted to write the results out to a new file, you would use the following lines of code.

```
f = open('/path/output_file_name.f999x999', 'wb')  # 'wb' means: write binary
new_data_array.tofile(f)
f.close()
```

NOTE that you could make any name you want for an output file name and note also that in Unix suffixes have no special meaning at all. I choose the end my file name with f999x999 because I know from working with the data that it is floating point numbers and that is a 2D array of 999 by 999 values AND because at some point in the future (possibly the distant future) when I want to read this data back into a new program, I need to know that it is floating point (hence the ***f***) and I need to know the dimensions are 999 by 999 elements (hence the ***999x999***).

READING ASCII FILES

1. Open a new text file and save as ***read_ascii_program1.py*** in your ***tutorial_exercises*** directory.
   NOTE: in the call to the function *open* in this example, the 'r' parameter indicates that we are opening this file for reading. A 'w' would indicate writing and 'a' for appending. If no parameter is supplied, as in the previous example, 'r' for reading mode is the default

```
ifile= '~/data/tutorial_data/smi_order.txt'
f=   open(ifile, 'r')
all_lines= f.readlines()          # reads in the entire ascii file as one very long stream
f.close()                         # of continuous ascii characters as a list
print(all_lines)                    #prints the list, including the \n characters

all_lines = ''.join(all_lines)   #converts the list to a string, with elements separated by ''
print(all_lines)                    #prints the string, with \n appearing as a new line
```

NOTE: When you read in all lines from an ascii file, each individual line ends with a *newline* character ('**\n**') and in most cases this character will need to be stripped off of each line with the ***.strip()*** function : e.g., **myline.strip() <--- This is a Pretty Big Deal**

2. The following is an example that would be used to write a text file out to the hard drive
   Note: write to file with the following functions:
   f.writelines(*List* of text lines) or f.write(single ascii string)

```
ofile= '~/data/tutorial_data/my_test_output.txt'
f=  open(ofile, 'w')
f.write('The quick brown fox\n')
f.write(' jumped over the\n')
f.write('lazy dog.')
f.close()
```

   Results in the contents of myfile.txt
   The quick brown fox jumped over the
   lazy dog.

READING HDF AND NETCDF FILES

All of the satellite data products you download from the web will be in HDF or netCDF format. This is a super nice format to use IF you have some nice functions to work with this data format. And fortunately for you I have written three ***VERY USEFUL*** hdf/netCDF functions that you are going to like a lot! These functions are:

- hdf_prod_info
- hdf_prod_scale
- read_hdf_prod

<------ These Three Functions are a **VERY BIG DEAL!**

Two important background points about HDF and netCDF files

1. HDF and netCDF files can contain a lot of different data products and data types in a single file. To read in a given data product from the file **you need to know the *name* of the data product you want to extract from the file**.

2. Each individual data product stored in an HDF or netCDF file **can have its own unique scaling and masking** associated with it that should be applied to the data right after it has been read from the file into the python program.

With points 1 and 2 in mind, the following steps need to be taken first when working with an HDF or netCDF file that are NEW to you (*i.e., a file that you do not know the names of the products inside and/or you do not know for sure if any scaling or masking needs to be applied after you read the file into your python program*).

Note: If you download a bunch of netCDF files of the same type (e.g., all MODIS-Aqua files from a single data order), you only need to check the product names and the scaling / masking info **on a single file** *because all the files will have the same basic product names and scaling / masking information*. **For this reason,** you should check the product names and scaling information of the product on just one file using python in ***Interactive Mode***.

After you have the product name and the scaling you will use this information (hard code the product name and scaling / masking) inside of a python script that reads in the data product and, if needed, applies the scaling / masking.

The following is an example of how this works….

1. Open a Unix Terminal window and ***cd*** to ~/data/tutorial_data
2. Type ***ls*** to see the file (and directories)
3. You should notice a file named: **A2010241173000.L2_LAC_OC.x.hdf**
4. Now launch python
5. Import into python the module that contains the HDF/netCDF functions I have written for this class by typing:

   from my_hdf_cdf_utilities import *

6. To find the names of the products contained in **A2010241173000.L2_LAC_OC.x.hdf** , type the following:

   fname = 'A2010241173000.L2_LAC_OC.x.hdf'
   hdf_prod_info(fname)

7. You should see a listing of all the products contained in the file printed to your monitor and among then should be the product: chlor_a (this is shorthand for chlorophyll a).

8. To see if any scaling needs to be applied to the 'chlor_a' product or if any fill values (i.e. bad data) need to be masked out by setting them to NaN, **type the following**:

   hdf_prod_scale(fname, 'chlor_a')

9. You should see a bunch of information about *chlor_a* printed to your monitor. In this simple case you will see that the ***slope/gain_factor*** = 1 and the ***intercept/add_offset*** = 0. So, in this case you do not need to apply any scaling correction. You may also see a fill value that needs to be masked (for example fill_value=-32767.0). In that case, you need to use the where statement to find the locations of all the fill values in the data array and set those locations to NaN. **You should always use this function to check for scaling and masking information whenever you read in a product**. Alternatively, you can look at the function **read_hdf_prod** in the library **my_hdf_cdf_utilities** to see that the parameter **nc_autoscale** is set to False by default. If you would like the read function to automatically apply scaling and masking information to your product, then you can send this optional parameter as in this example:

   read_hdf_prod(fname, product_name, nc_autoscale=True)

   Keep in mind that this will only affect the automatic masking and scaling of netCDF files but will have no effect on hdf files.

Now that you have determined the product name and checked about any scaling / masking information, you can confidently use this information in any future programs you write that reads in this kind of file (in this case a MODIS-Aqua files).

---

ncdump -h

There is a very powerful netCDF function that is found on most computers called **ncdump**

Typing: **ncdump -h** *the_name_of_your_netcdf_file* at the unix terminal (not within python) will dump all of the header information contained in the file. This will included (among a lot of other information) the product names in the file and the scaling information.

Typing: **ncdump** *the_name_of_your_netcdf_file* at the unix terminal (not within python) will dump all of the information contained in the file – which means the header information and the data itself. You don't want to do this. You really only want to see the header information so be sure to use **ncdump -h**.

The next steps show how you would use the information above (*obtained just once*) in a python program that loops through and reads in a bunch of HDF/netCDF files…

---

10. ***cd*** back to your ***~/python_programs/tutorial_exercises*** directory

11. Open a new *text file* and save as ***read_hdf_cdf_program1.py*** in your ***tutorial_exercises*** directory

```
#!/usr/bin/env python

from my_hdf_cdf_utilities import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors

fname= '~/data/tutorial_data/ A2010241173000.L2_LAC_OC.x.hdf'
prodname= 'chlor_a'
slope= 1.0                #Found using hdf_prod_scale(fname,prodname)
intercept= 0.0            #Found using hdf_prod_scale(fname,prodname)
fill_value=-32767.0       #Found using hdf_prod_scale(fname,prodname)

chl_array = read_hdf_prod(fname, prodname)  # read the product from the hdf/netcdf file
bad_loc=np.where(chl_array == fill_value)     #Search for fill values before applying scale
chl_array = slope*chl_array + intercept
chl_array[bad_loc]=np.nan          # NaN out fill values after applying scale in case
                                   # product is an integer type before multiplying
                                   # it by a floating point slope value
print(chl_array.shape)
```

12. **Note:** If you feel like it, you can display chl_array using the same approached given above under the section: "**imaging 2D binary data to the monitor"**

13. Run your program from the Unix Terminal Window by typing: ***python read_hdf-cdf_program1.py***

14. NOTE (*future reference*): To write any 2D array out as a ***netCDF*** file, I have written a very simple generic write function that outputs 2D data to a file of your own naming. The output product name will be: '***data***'. Below is a hypothetical example of how it would be used.

    ofname= '~/data/my_new_outputfile.nc'
    write_generic_2D_netcdf4(ofname, chl_array)

MAKING SIMPLE X-Y PLOTS WITH PYPLOT MODULE

Simple plot (and more elaborate ones too) can be done using the **pyplot** *module* found in the **matplotlib** *Package*. The following exercise gives a simple example.

1. Open a new text document called **myprogram2.py** and save it in your tutorial_exercises directory.
2. Add the following lines of code to your text document…

```
import numpy as np
import matplotlib.pyplot as plt

#see: http://matplotlib.org

year= np.array(range(16))+2000    # make up a year vector going from 2000 to 2015
chl= np.random.rand(16)           # make up fake chlorophyll data to plot again year
                                  # chl is a series of random numbers between 0 and 1.
plt.plot(year, chl)
plt.ylabel('annual chlorophyll')
plt.xlabel('year')
plt.show()
plt.close()
```

3. Save the content of the text file
4. Open a Terminal Window and cd to your ~/python_programs/tutorial_exercises directory and run your new python program by typing***: python myprogram2.py***

IF STATEMENTS

Up until now, all of the code you have written has been processed linearly in order line-by-line. But in larger programs, your script will need to jump all over the place, accessing other functions, running through loops, and operating differently depending on the validity of a conditional expression. The means you will be seeing a lot of two new programming tools:

1) the IF statement and 2) the FOR LOOP

Python uses **Indentation to signify a *block* of statements** within an ***IF*** condition or within a ***FOR*** Loop. Although this makes your programs much easier to read, it does mean that *Python is Whitespace Sensitive*, so ***be cautious using tabs*** in your script – it is usually better to just the spacebar on the keyboard.

NOTE: When indenting blocks of statements of statements within an IF condition or FOR it is considered good coding form to use ***4 white spaces to form your indented block of statements***.

NOTE: The line of code that sets up the IF or FOR conditions **always ends with a colon (*:*)**

Although you can write *if*-statements in the interactive mode of Python, it is easier to edit, and make changes to them, if you write them in a text file (*a python script*) and run them with python in *Interpreter Mode.*

The following exercise will give you an example of how the *if*-statement works in a python script.

1. Use *Sublime* to create a new text file called ***myprogram3.py*** and save it in ***~/python_programs/tutorial_exercises***
2. Write the following lines of code into the open text window…

```
#!/usr/bin/env python

x = -2
if x > 0:
   print('x is > 0')
elif x < 0:
   print('x is < 0')
else:
   print('x is equal to 0')
```

3. Save the text files to your ***tutorial_exercises*** directory that was created earlier.
4. Open a Unix Terminal Window and cd to your ***tutorial_exercises*** directory
5. **Type: *ls*** to see a listing of the files in your directory
6. Run your new python program by **typing: *python myprogram3.py***

Note: ***elif* is short for "else if"** and is only executed if the first *if* statement above it is false. *The else* is only executed if both statements above it (the if and elif) are false. Play around with the value of x to see how the program behaves.

FOR LOOPS

There are two kinds of FOR loop indexing in Python:

- A traditional FOR loop sequentially increments a numerical value to index a *Tuple, List, Array,* or *String*. For example, a loop can sequentially increment *i* from 0 to 10 in a for loop that has something like ***my_array[i]*** inside the loop…

- Python also has *For-Each* loops in which the iterator can be the *values contained* within a tuple, list, array, or string. Collectively, tuples, lists, arrays, and strings are called iterables because they can be iterated over.

The following is a simple exercise Traditional *For* loop in Python and the *For-Each* loop. Note the use of the Python range() function, which creates a *List* of integer values using the syntax: ***range(start, stop, step)***. If no start is included, Python assumes 0 by default. If no step is included, Python assumes 1 by default.

NOTE: Recall that a block of statements within a FOR loop is designated with ***indentation***.

1. Use *Sublime* to create a new text file called ***myprogram4.py*** and save it in ***~/python_programs/tutorial_exercises***

2. Write the following lines of code into the open text window…

```
x = [3.4, 2, 9.8, 0]

print('the list x is given by: ',  x)
print('the list given by range(len(x)) is:  ', range(len(x)))
print(' ')

for i in range(len(x)):
    x[i] = x[i] + 1

print('the new list x after the loop is completed is given by: ' , x)
```

1. Save the text files to your ***tutorial_exercises*** directory that was created earlier.
2. Open a Unix Terminal Window and cd to your ***tutorial_exercises*** directory
3. **Type: *ls*** to see a listing of the files in your directory
4. Run your new python program by **typing: *python myprogram4.py***

There are plenty of situations in which you do not need to know the indices of elements in an iterable. For example, if you want to loop through **a list of filenames**, You can use the following example:

1. Use *Sublime* to create a new text file called ***myprogram5.py*** and save it in ***~/python_programs/tutorial_exercises***

2. Write the following lines of code into the open text window…

```
filename_list = ['file1.txt', 'file2.txt', 'file3.txt']

for name in filename_list:
   print(name)
```

3. Save the text files to your ***tutorial_exercises*** directory that was created earlier.
4. Open a Unix Terminal Window and cd to your ***tutorial_exercises*** directory
5. **Type: *ls*** to see a listing of the files in your directory
6. Run your new python program by **typing: *python myprogram5.py***

BEST PRACTICES FOR ORGANIZING LARGE PROGRAMS INTO A *MAIN PROGRAM* THAT CALLS A SERIES OF *EXTERNAL SUB-PROGRAMS*…

Up until now you have written blocks of code directly into your scripts. But what if you want to run that same block of code again, or from a different program? Instead of copying and pasting all of this code every time you want to reuse it in another program, you can create a *Method* that can then simply be *imported* into any new programs you write.

There are 3 types of Methods: **functions, procedures, and constructors**. You can create as many methods as you want in a single Python script. In general, *Functions* return values (a single value or a Tuple of values or a List of values or an Array of values). *Procedures* don't return anything, and *Constructors* are used to create objects. ***Constructors and Classes are not covered in this course***.

- Both functions and procedures begin with the ***def*** keyword to denote the start of a given method
- Both use the same *Indentation* and *Colon (:)* syntax that was used for the *IF* statements and *FOR* loops
- The only difference between a function and a procedure is that a function will have a **return** statement in the code.

The following exercise will provide you with an example. *By the way, when you start running the python scripts that I have written for ocean color processing you will see lots of methods contained in text files I have created which have a long list of procedures.*

Create a new text file called ***mymethods.py*** and save it in your ***tutorial_exercises*** directory.

1. Type out the following lines of code into the open text file.

```
#!/usr/bin/env python

# function to list only filenames that have 2003 in the name.
def myfunction(all_fnames):
    results = []                    #creates an empty list called results
    for fname in all_fnames:
        if '2003' in fname:
            results.append(fname)
    return results

# Procedure to print the first n numbers in the Fibonacci Sequence
def myprocedure(n):
    a, b = 0, 1                     # assigns 0 to a and 1 to b
    for i in range(n):
        print(a)
        a, b = b, a + b             # assigns b to a and b + a to b
```

3. Save the text file.
4. Go to the Unix terminal window and ***cd*** to your ***tutorial_exercises*** directory
5. Launch python from your terminal window by **typing: *python***
6. Import the functions and procedures you created by typing:

   >>> ***import mymethods***

7. You now have access to all of the functions and procedures contained in mymethods.py. Type the following to see how you can use them with python in ***Interactive*** *Mode*.

   ```
   >>> all_fnames = ['S1999.f99x99', 'A2003.f34x34', 'H2014.f62x49', 'S2003.f45x54']
   >>> fnames = mymethods.myfunction(all_fnames)
   >>> print(fnames)
   >>> mymethods.myprocedure(12)
   ```

Note: The same lines of code above can be placed into a new text file and then called with python in ***Interpretive*** *Mode* as followings (optional exercise):

8. Create a new text file called ***myprogram6.py*** and place the following in the open text widow:

   ```
   import mymethods
   all_fnames = ['S1999.f99x99', 'A2003.f34x34', 'H2014.f62x49', 'S2003.f45x54']
   fnames = mymethods.myfunction(all_fnames)
   print(fnames)
   mymethods.myprocedure(12)
   ```

9. Save ***myprogram6.py***
10. Open a Unix Terminal Window and cd to your ***tutorial_exercises*** directory
11. **Type: *ls*** to see a listing of the files in your directory
12. Run your new python program by **typing: *python myprogram6.py***

Note: *You will soon see a lot of python scripts that use the* ***import*** *statement at the top of the program to import a wide variety of pre-written python functions and procedures!!!*

SOME BASIC THINGS TO DO WITH YOUR 2D ARRAYS (I.E., SATELLITE IMAGES)

COMPOSITE-AVERAGING 2D IMAGE DATA

There are going to be **TONS** of times when you will want to composite-average 2D images to form a new image composed of the element-by-element average. One of the main reasons for composite-averaging is to combine a bunch of daily images that that have a lot of missing data because of clouds and make a weekly or monthly (or whatever time period you want) average composite image with most of the clouds filled in with data from the combined daily data.

The next two examples demonstrate how to make a composite-average. The first example is simple and the second is elegant and even necessary in the case of averaging daily data into a yearly composite average.

A SIMPLE COMPOSITE-AVERAGE APPROACH

1. Open a new text file and save as ***myprogram7.py*** in your ***tutorial_exercises*** directory and add the following lines of code to the new text file…

```
#!/usr/bin/env python
import numpy as np                    # imports all of the numpy methods
import matplotlib.pyplot as plt
import matplotlib.colors

file1 = '~/data/tutorial_data/S1998148172338_chlor_a.f999x999'
file2 = '~/data/tutorial_data/S1998149163038_chlor_a.f999x999'

# open binary files and save data into arrays
f1 = open(file1)
data1 = np.fromfile(f1, dtype=np.float32)   #assumes data were previously written out to
f1.close()                                  #hard drive as 32bit floating point numbers

f2 = open(file2)
data2 = np.fromfile(f2, dtype=np.float32)   #assumes data previously written out to
f2.close()                                  #hard drive as 32bit floating point numbers

# reshape data into 2D matrices
data1 = data1.reshape([999, 999])   #you would have to know a priori that the data
data2 = data2.reshape([999, 999])   #had previously been written out as a 999x999 array
```

> **Note:** For the next step you would have to know *a priori* that the data being read in from the hard dive above used not-a-number (***NAN***) values to designate missing data locations (most often because cloud cover prevented the collection of good data to compute chlorophyll concentration).

2. Now add the following lines to ***myprogram7.py*** to average the two

```
# average arrays
total_array  = np.nan_to_num(data1) + np.nan_to_num(data2)
total_counts = (~np.isnan(data1)).astype(int) + (~np.isnan(data2)).astype(int)
avg_array = total_array/total_counts
```

- **np.nan_to_num** replaces locations that have *NANs* with *Zeros* and infinities with large numbers.
- **np.isnan** returns an array the same size as data1 or data2 with TRUE values where there are NANs and FALSE values where there are real numbers. **However**, the tilde (~) operator is shorthand for invert(), which is the bit-wise inversion operator, which means it switches 1's to 0's and switches 0's to 1's. Since the binary value of TRUE is 16 ones and the binary value of FALSE is 16 zeroes, the ~ acts to flip the TRUE to a FALSE value, and flip the FALSE value to a TRUE value. So **~np.isnan** reverses the original meaning of np.isnan and returns FALSEs where there are NANs and TRUEs where there are real numbers.

- **astype()** makes a copy of an array and casts it as the variable type passed in the

parameters. In this case, we converted the array of Boolean (TRUE and FALSE values) to the integer variable type **int.** So TRUE becomes 1 and FALSE becomes 0.

To better understand ~np.isnan() function, try the following code in *interactive mode*:

```
>>>> import numpy as np
>>>> a=np.array([1.0,2.0,3.0])
>>>> print(a)
>>>> a[1]=np.nan
>>>> print(a)
>>>> np.isnan(a)
>>>> ~np.isnan(a)
>>>> (~np.isnan(a)).astype(int)
```

- Finally, *numpy* allows you to add, subtract, multiply and divide 2D arrays element-by-element so for a general example

  avg_array= (data_asrray1 + data_array 2)/(counts_array1 + counts_array2)

  The expression above would do the math separately for each individual element in the 2D arrays to produce an avg_array that is the same size as the data and count arrays with the average properly computed for each array element location.

3. Using the **"display 2D images to the monitor**" method described earlier, display the each of the three 2D data arrays (i.e., data1, data2 and avg_array).
   NOTE: Create all three figures before calling the show() function if you want to see all three figures at once

4. Run your program from the Unix Terminal Window by typing: ***python myprogram7.py***

   NOTE: You will probably get a message like "RuntimeWarning: invalid value encountered in divide". Just ignore this. It a warning and not an error. It says that python made some NaNs when it divided by zero – which is what you want.

A MUCH MORE ELEGANT COMPOSITE-AVERAGE APPROACH…

NOTE ---> You will do this for a homework assignment later, so I will not use actual data files here, but the basic steps are shown here for Looping through a long series of files and average them one-by-one into a single final composite-average. **The code looks something like this…**

Note: *You will see near the end of this handout that* ***glob.glob*** *is a function that returns a list of file names in a user-specified directory…*

```
#!/usr/bin/env python
import numpy as np
import glob

# search for files in ~/data/tutorial_data that have 1998 as some part of the file name.
filename_list= glob.glob('~/data/tutorial_data/S1998*')



xdim=999
```

```
ydim=999

cuml_data_2d=     np.zeros((xdim, ydim))
cuml_counts_2d= np.zeros((xdim, ydim))

for name in filename_list:

    f = open(name)
    data_2d = np.fromfile(f, dtype=np.float32)
    f.close()
    data_2d = data_2d.reshape([ydim, xdim])

    cuml_data_2d =      cuml_data_2d + np.nan_to_num(data_2d)
    cuml_counts_2d =   cuml_counts_2d + (~np.isnan(data_2d)).astype(int)

average_data_2d= cuml_data_2d/ cuml_counts_2d
```

SUBSCENE SMALLER IMAGES FROM LARGER IMAGES

1. Open a new text file and save as ***myprogram8.py*** in your ***tutorial_exercises*** directory and add the following lines of code to the new text file…

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors

filename = '~/data/tutorial_data/S1998148172338_chlor_a.f999x999'

f = open(filename)
full_image = np.fromfile(f, dtype=np.float32)
f.close()
full_image = full_image.reshape([999, 999])

sub_image = full_image [300:700, 100:400]
```

15. Using the same approached given above under the section: "**imaging 2D binary data to the monitor"** under the section for reading in numerical binary data, display both the starting full_image and the sub_image.

16. Run your program from the Unix Terminal Window by typing: ***python myprogram8.py***

*It is **SUPER IMPORTANT** to note that the ordering of the two dimensions of any 2D array in python is (row, column) or equally (y,x) – this might be the opposite ordering from other programming languages you might have experience with.*

DRAW A COASTLINES FOR PNG IMAGE OUTPUT

A really useful toolkit that allows you to plot coastlines and plot geographic data on various projections is a matplotlib in conjunction with cartopy. The following website includes example code for plotting.
See this for more info: https://scitools.org.uk/cartopy/docs/v0.15/matplotlib/intro.html

NOTE: *In this example someone (me) has already crated a mapped satellite data product (a 2d array of data) using a known map projection and know latitude and longitude boundaries. It is assumed that you somehow were already given the projection and lat/lon formation beforehand.*

1. Open a new text file and save it as ***myprogram9.py*** in your ***tutorial_exercises*** directory and add the following lines of code to the new text file…

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors
import matplotlib.colorbar

import cartopy.crs as ccrs
from cartopy.mpl.ticker import LongitudeFormatter, LatitudeFormatter
import cartopy

fname = '~/data/tutorial_data/S1998148172338_chlor_a.f999x999'
f = open(fname)
mapped_data = np.fromfile(f, dtype=np.float32)
f.close()
mapped_data = mapped_data.reshape([999, 999])

# set the map lat/lon bounds (must know this a priori)
# ---
north= 46.0
west= -72.0
south= 37.0
east= -63.0
map_data_extent=[west, east, south, north]

# Set map projection space for the plot window...
# Note: you must know the map projection and lat/lon limits a priori...
# --
# Set Center Longitude Value (lon_ctr) value: Choices are 0 or 180
# --> Use lon_ctr=0 for map longitude limits that do not cross the international dateline.
#       (use -180 to +180 longitude convention).
# --> Use lon_ctr=180 for map longitude limits that international dateline.
#       (use 0 to 360 longitude convention).
# see: https://stackoverflow.com/questions/13856123/setting-up-a-map-which-crosses-the-dateline-in-cartopy
# ---

lon_ctr=0.0



ax= plt.axes(projection=ccrs.PlateCarree(central_longitude=lon_ctr))
```

```
ax.set_extent(map_data_extent, crs=ccrs.PlateCarree())

# set the color palette
mycmap= plt.get_cmap('nipy_spectral').copy()
mycmap.set_bad('k')


# Image the satellite data, add color table and min/max values and the add a color bar.
#
# Note: --> use the "norm=matplotlib.colors.LogNorm(##, ##)" keyword in ax.imshow()
# below for  the case of lognornal data (e.g. chloro_a) -- omit this keyword for linear
# data (e.g., sst) and instead use vmin= ##, vmax= ##.  Where in both cases ## are the
# numerical values of the minimum and maximum of your data range.
# ---
sat_img= ax.imshow(mapped_data, transform=ccrs.PlateCarree(), \
extent=map_data_extent, cmap=mycmap, norm=matplotlib.colors.LogNorm(0.01, 20.0))
plt.colorbar(sat_img)

# Fill continents with gray and use black for edge (coastline)...
# ---
feature = cartopy.feature.NaturalEarthFeature(name='land', category='physical', scale='10m',\
edgecolor='k', facecolor='gray')
ax.add_feature(feature)

# Set up Lat/Lon Labels...
# ---
meridians = np.arange(west, east,2.)
parallels = np.arange(south, north,2.)
# ---
ax.set_xticks(meridians, crs=ccrs.PlateCarree())
ax.set_yticks(parallels, crs=ccrs.PlateCarree())
# ---
lon_formatter = LongitudeFormatter(zero_direction_label=True)
lat_formatter = LatitudeFormatter()
# ---
ax.xaxis.set_major_formatter(lon_formatter)
ax.yaxis.set_major_formatter(lat_formatter)


# Save the mapped images as a png file…
plt.savefig("~/data/test.png", bbox_inches='tight')
# NOTE--> you might have to explicitly write your home directory (~/) as /Users/username)

# Show the plot to the monitor…
plt.show()

# Close plotting function and clear all plotting settings…
plt.close()
```

2. Run your program from the Unix Terminal Window by typing: ***python myprogram9.py***

NOTE: You will have to click on the image window that appears on the monitor to close the image before your python program continues and executes the png save function. *---It is due to*

a quirky thing about imshow().

FILE SEARCH (GLOB.GLOB)

Glob.Glob --- *Read in a list of filenames from a specified directory:*
There is a Python function called ***glob*** that is used **A LOT** to read in a list of file names from a specified directory.
Normally you would use this in a written python program, but you can run it from the python Interactive Mode to see clearly how it works.

1. Launch python from a Terminal Window and then type the following

```
>>> import glob
>>> filename_list= glob.glob('~/data/tutorial_data/*1998*')
>>> print(filename_list)
```

Note: glob is the name of a module and inside that module there is a function also called glob. So, to use the function in the module (i.e. using the ***module.function*** naming convention) you have ***glob.glob***.

The above code will read in all file names that have *1998* somewhere in the file name itself. The result (filename_list) is a python *List* of the file names there were found in the directory: ~/data/tutorial_data

NOTE --> *Using **glob.glob** for reading in a list of file names is... **A Really Big Deal**!!*

CALLING OUT TO THE UNIX COMMAND LINE ENVIRONMENT

To run Unix Level commands from within a python program (e.g., ***mkdir*** or **rm**) just like you would in the Terminal Window, you have two options. You can use the **os.system** command, or the **subprocess.call** command. The **os.system version is simpler**, but **subprocess.call gives you more power** by allowing you to access and to use standard Unix input and output and error streams - meaning you can take into your python program the information output from a given Unix command made with **subprocess.call**.

When you don't care about the output:

```
import subprocess
cmd = 'python ~/python_programs/tutorial_exercises/myprogram9.py'
subprocess.call(cmd, shell=True)
```

When you do care about the output:

```
import subprocess
cmd = 'ls  ~/python_programs'
pipe = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE).stdout
my_text_from_ls_command = pipe.read().strip()
pipe.close()
print(my_text_from_ls_command)
```

SOME HANDY UNIX COMMANDS

There are lots and lots of books written on the Unix operating system with many of them hundreds of pages long. But you can get along pretty well as a beginner with just a handful of Unix commands. Below is a list of the most basic commands that are use A LOT!

pwd	***p***rint ***w***orking ***d***irectory path
cd dir1	***c***hange to ***d***irectory dir1
ls	***l****i****s***t file names (and/or subdirectories) contained in the current working directory
cp fname1 fname2 cp fname1 path/	make a ***c****o****p***y of fname1 and call it fname2 make a ***c****o****p***y of fname1 and place it in a new directory described by path/
cp -r dir1 path/	***c****o****p***y ***r***ecursively dir1 (*and all contained subdirectories*) and place it in a new directory described by path/
mkdir dir	make a new directory
rm fname	remove (delete) fname
rmdir dir1	remove *empty* directory
rm -r dir1	remove (recursively) a directory and everything inside the directory

Two Key Unix Concepts…

1. Unix distinguishes between *upper* and *lower* case letters.

2. A Unix file name consists of two components (path and base name)…

   filename = ***directory path*** + **base name**

   - For example: /dir1/subdir2/subdir3/fname1.txt.
     - ***The path*** is: /dir1/subdir2/subdir3
     - ***The base name*** is: fname1.txt

A HANDY JULIAN DAY <–> CALENDAR DATE CONVERSION TABLE

Satellite filenames are often written with Julian day as part of the filename. If you want to select files for a given month it can help to convert between Julian day and calendar date. Here is a handy table to make the conversion

Julian Day Converter													
Jday	Jan-Feb	Jday	Mar-Apr	Jday	May-Jun	Jday	Jul-Aug	Jday	Sep-Oct	Jday	Nov-Dec
1	1/1/03	60	3/1/03	121	5/1/03	182	7/1/03	244	9/1/03	305	11/1/03
2	1/2/03	61	3/2/03	122	5/2/03	183	7/2/03	245	9/2/03	306	11/2/03
3	1/3/03	62	3/3/03	123	5/3/03	184	7/3/03	246	9/3/03	307	11/3/03
4	1/4/03	63	3/4/03	124	5/4/03	185	7/4/03	247	9/4/03	308	11/4/03
5	1/5/03	64	3/5/03	125	5/5/03	186	7/5/03	248	9/5/03	309	11/5/03
6	1/6/03	65	3/6/03	126	5/6/03	187	7/6/03	249	9/6/03	310	11/6/03
7	1/7/03	66	3/7/03	127	5/7/03	188	7/7/03	250	9/7/03	311	11/7/03
8	1/8/03	67	3/8/03	128	5/8/03	189	7/8/03	251	9/8/03	312	11/8/03
9	1/9/03	68	3/9/03	129	5/9/03	190	7/9/03	252	9/9/03	313	11/9/03
10	1/10/03	69	3/10/03	130	5/10/03	191	7/10/03	253	9/10/03	314	11/10/03
11	1/11/03	70	3/11/03	131	5/11/03	192	7/11/03	254	9/11/03	315	11/11/03
12	1/12/03	71	3/12/03	132	5/12/03	193	7/12/03	255	9/12/03	316	11/12/03
13	1/13/03	72	3/13/03	133	5/13/03	194	7/13/03	256	9/13/03	317	11/13/03
14	1/14/03	73	3/14/03	134	5/14/03	195	7/14/03	257	9/14/03	318	11/14/03
15	1/15/03	74	3/15/03	135	5/15/03	196	7/15/03	258	9/15/03	319	11/15/03
16	1/16/03	75	3/16/03	136	5/16/03	197	7/16/03	259	9/16/03	320	11/16/03
17	1/17/03	76	3/17/03	137	5/17/03	198	7/17/03	260	9/17/03	321	11/17/03
18	1/18/03	77	3/18/03	138	5/18/03	199	7/18/03	261	9/18/03	322	11/18/03
19	1/19/03	78	3/19/03	139	5/19/03	200	7/19/03	262	9/19/03	323	11/19/03
20	1/20/03	79	3/20/03	140	5/20/03	201	7/20/03	263	9/20/03	324	11/20/03
21	1/21/03	80	3/21/03	141	5/21/03	202	7/21/03	264	9/21/03	325	11/21/03
22	1/22/03	81	3/22/03	142	5/22/03	203	7/22/03	265	9/22/03	326	11/22/03
23	1/23/03	82	3/23/03	143	5/23/03	204	7/23/03	266	9/23/03	327	11/23/03
24	1/24/03	83	3/24/03	144	5/24/03	205	7/24/03	267	9/24/03	328	11/24/03
25	1/25/03	84	3/25/03	145	5/25/03	206	7/25/03	268	9/25/03	329	11/25/03
26	1/26/03	85	3/26/03	146	5/26/03	207	7/26/03	269	9/26/03	330	11/26/03
27	1/27/03	86	3/27/03	147	5/27/03	208	7/27/03	270	9/27/03	331	11/27/03
28	1/28/03	87	3/28/03	148	5/28/03	209	7/28/03	271	9/28/03	332	11/28/03
29	1/29/03	88	3/29/03	149	5/29/03	210	7/29/03	272	9/29/03	333	11/29/03
30	1/30/03	89	3/30/03	150	5/30/03	211	7/30/03	273	9/30/03	334	11/30/03
31	1/31/03	90	3/31/03	151	5/31/03	212	7/31/03	274	10/1/03	335	12/1/03
32	2/1/03	91	4/1/03	152	6/1/03	213	8/1/03	275	10/2/03	336	12/2/03
33	2/2/03	92	4/2/03	153	6/2/03	214	8/2/03	276	10/3/03	337	12/3/03
34	2/3/03	93	4/3/03	154	6/3/03	215	8/3/03	277	10/4/03	338	12/4/03
35	2/4/03	94	4/4/03	155	6/4/03	216	8/4/03	278	10/5/03	339	12/5/03
36	2/5/03	95	4/5/03	156	6/5/03	217	8/5/03	279	10/6/03	340	12/6/03
37	2/6/03	96	4/6/03	157	6/6/03	218	8/6/03	280	10/7/03	341	12/7/03
38	2/7/03	97	4/7/03	158	6/7/03	219	8/7/03	281	10/8/03	342	12/8/03
39	2/8/03	98	4/8/03	159	6/8/03	220	8/8/03	282	10/9/03	343	12/9/03
40	2/9/03	99	4/9/03	160	6/9/03	221	8/9/03	283	10/10/03	344	12/10/03
41	2/10/03	100	4/10/03	161	6/10/03	222	8/10/03	284	10/11/03	345	12/11/03
42	2/11/03	101	4/11/03	162	6/11/03	223	8/11/03	285	10/12/03	346	12/12/03
43	2/12/03	102	4/12/03	163	6/12/03	224	8/12/03	286	10/13/03	347	12/13/03
44	2/13/03	103	4/13/03	164	6/13/03	225	8/13/03	287	10/14/03	348	12/14/03
45	2/14/03	104	4/14/03	165	6/14/03	226	8/14/03	288	10/15/03	349	12/15/03
46	2/15/03	105	4/15/03	166	6/15/03	227	8/15/03	289	10/16/03	350	12/16/03
47	2/16/03	106	4/16/03	167	6/16/03	228	8/16/03	290	10/17/03	351	12/17/03
48	2/17/03	107	4/17/03	168	6/17/03	229	8/17/03	291	10/18/03	352	12/18/03
49	2/18/03	108	4/18/03	169	6/18/03	230	8/18/03	292	10/19/03	353	12/19/03
50	2/19/03	109	4/19/03	170	6/19/03	231	8/19/03	293	10/20/03	354	12/20/03
51	2/20/03	110	4/20/03	171	6/20/03	232	8/20/03	294	10/21/03	355	12/21/03
52	2/21/03	111	4/21/03	172	6/21/03	233	8/21/03	295	10/22/03	356	12/22/03
53	2/22/03	112	4/22/03	173	6/22/03	234	8/22/03	296	10/23/03	357	12/23/03
54	2/23/03	113	4/23/03	174	6/23/03	235	8/23/03	297	10/24/03	358	12/24/03
55	2/24/03	114	4/24/03	175	6/24/03	236	8/24/03	298	10/25/03	359	12/25/03
56	2/25/03	115	4/25/03	176	6/25/03	237	8/25/03	299	10/26/03	360	12/26/03
57	2/26/03	116	4/26/03	177	6/26/03	238	8/26/03	300	10/27/03	361	12/27/03
58	2/27/03	117	4/27/03	178	6/27/03	239	8/27/03	301	10/28/03	362	12/28/03
59	2/28/03	118	4/28/03	179	6/28/03	240	8/28/03	302	10/29/03	363	12/29/03
		119	4/29/03	180	6/29/03	241	8/29/03	303	10/30/03	364	12/30/03
		120	4/30/03	181	6/30/03	242	8/30/03	304	10/31/03	365	12/31/03
						243	8/31/03				

NOTE: Make a new directory **under the main python_programs directory** called: ***python_problem_set*** and save each of the following problem programs in this directory…

PROBLEM 1 – The SST data contained in the file named: ***/Users/YourUsername/data/tutorial_data*/sst/2004.mon06.f774x843** is a monthly averaged SST data image for the month of June 2004 (*with NANs used for missing data*). It has geographic boundary coordinates: **Latitude = 25 to 62 and Longitude = -82 to -48** (i.e., NW Atlantic) and there are **774x843 *floating-point* array elements** (or equivalently *pixels*) in the image. The image is mapped in ***cylindrical coordinates,*** so each pixel represents a fixed fractional number of degrees of latitude and longitude.

Write a python script called **problem1.py** that will let you input a latitude and longitude into your program at the python command line and have the program print out the corresponding SST value found at that location.

(*See **Hints to Problem 1** for a "hint" on how to get started with this problem*).

1. Start with a new text file (save it as *problem*1.py) and write:

```
#! /usr/bin/env python
import numpy as np

lat= float(input('\nenter your latitude here: '))
lon= float(input('enter your longitude here: '))

… read in sst…                    #see tutorial for reading in numerical binary data
… covert lat to a row_value…      #see attached hint to this problem
… convert lon to a column_value…  #see attached hint to this problem

print(sst[row_value, col_value])
```

2. The code reads in the SST data to a 2D array, navigates to a correct pixel (array element) location for any prescribed latitude-longitude pair coordinate pair.
3. Once the procedure is written, you need to open a *Terminal Window,* ***cd*** to *python_problem_set* directory and then **type: *python problem1.py***

SEE HINTS TO THIS PROBLEM ON NEXT PAGE

HINTS TO GET STARTED WITH PROBLEM 1:

1. The latitude boundaries are 25 to 62 and the longitude boundaries are -82 to -48
2. The array dimensions are **843 rows** (y-dimension) by **774 columns** (x-dimension).
3. You need to be able to convert specified longitude and latitude values (***lat, lon***) into corresponding array element indices for the row and column direction (***irow, icol***).
4. Finally, you want to print the sst value contained in the sst array at array element locations irow, icol using the form: ***print(sst_array[irow,icol]***).

LONGITUDE CONVERSION –
The way I approach the problem is getting the expression that converts longitude values to array column index values (***icol***), to work properly for the extreme ends of the image. The approach requires that when you give the python program a longitude of -82 degrees, the corresponding computed column index (***icol***) value (element number) has to be 0 and when you give the python program a longitude of -48, it has to calculate an column index value (***icol***) of 773 (note: since the array is 774 elements wide and indexing starts at 0 so the last possible array column index is 773).

1. *The following expression will give you an x-array index value (**colx**) of zero when the given **lon** is equal to -82, which is what you want.*
   icol = (lon + 82.0).

   This expression satisfies the conversion problem for the left side of the image, but the right side of the image is still in need of some help.

The next step is to figure out how to get ***icol*** to increase linearly from zero to 773 as ***lon*** goes from -82 to -48. The key is to figure out how many pixels to the right (columns) you need to move in the image for every degree of longitude you want to move to the east. Keep in mind that you want to end with ***icol*** = 773 when ***lon*** is set to -48. Setting ***lon*** to -48 in the expression: **icol = (lon + 82.0),** yields delta 34 degrees which means you move a total of 34 degrees of longitude to get to the far right side of the image.

2. *Keeping in mind that **icol** must equal 773 when given_lon is -48, it will become apparent, with some thought, that the following expression will work for you:* ***icol = (lon + 82.0)*773.0/34.0***

When ***lon*** is -82, ***icol*** is zero. When ***lon*** is -48 ***icol*** is 773. Since the expression is linear, ***icol*** will increase linearly between 0 and 773, as ***lon*** is varied between -82 and -48.

LATITUDE CONVERSION –
The same approach (steps 1 and 2 above) is taken for the row index (***irow***). Keep in mind that python counts its indices from the top to the bottom, so you want to set up the expression such that when ***lat*** = 62, you want ***irow*** to be equal to zero and when ***lat*** is 25 you want ***irow*** to be equal to 842.

PROBLEM 2 – A very useful task in remote sensing is to **subset out a portion of a larger satellite image** to obtain just the geographic area of interest.

1. Write a new python script called ***problem2.py*** that start with the full image contained in the file used in Problem 1: ***/Users/YourUsername/data/tutorial_data*/sst/2004.mon06.f774x843** with known pixel size (**774x843**) and **latitude (25 to 62)** and **longitude (-82 to -48)** and subscenes out the Gulf of Maine region with **latitude 37 to 46** and **longitude –72 to -63**.

Note this problem is basically doing problem 1 all over again. In problem 1, it was a single_sst_value= sst_array[row_value, column_value] and in the present problem it is: **gulf_maine_sub_image= full_sst_array[*row1:row2, col1:col2*]**. The colons mean: take all pixels between ix1 and ix2 array elements and between iy1 and iy2 array elements.

2. Display both the main image and the subset image (*see python tutorial for imaging 2D data to the monitor*). Be sure to import *numpy* and *pylab* modules at the top of problem2.py

---

PROBLEM 3 – The bathymetry file: */Users/YourUsername/data/tutorial_data/bathymetry.i999x999* is a 999x999 ***integer*** array (i.e., **dtype=int16**) that has exactly the same geographic boundaries as the chlorophyll file: */Users/YourUsername/data/tutorial_data/S1998148172338_ chlor_a.f999x999* (i.e., the Gulf of Maine Region Bound by **latitude: 37 to 46** and **longitude: -72 to -63**).

1. Write a python procedure called problem3.py that will let you input a *minimum depth* and a *maximum depth* and then will print out a ***mean*** and ***standard deviation*** for the surface chlorophyll concentrations that overlay the regions with water column depth is between the given minimum and maximum bathymetric values. *Note: Ocean depths vary from 0 to about 4,000 meters.*

   Hint the numpy ***where*** command will be used with the bathymetry array to get *array element locations* for the desired bathymetry interval and then the result of the ***where*** command (i.e., the *Tuple* of row, column values for satisfied bathymetry *locations*) will be used in the chlorophyll array to select the chlorophyll concentration at the location overlying the same bathymetry interval – **this works because both arrays are exactly the same pixel size and both have the same lat/lon boundaries**.

   Note: Missing chlorophyll values are flagged by ***NAN***. To compute the mean and standard deviation you should use the following ***numpy functions***:

   - ***nanmean()***
   - ***nanstd***()

2. Start with a new text file called ***problem3.py*** and write:

```
#! /usr/bin/env python
import numpy as np
```

```
min_depth= float(input('\nenter minimum depth here: '))
max_depth= float(input('\nenter maximum depth here: '))

… read in bathymetry file…          #remember this is integer data
… read in chlorophyll file          #remember this is float32 data
… use numpy where to find locations of min-max depths

... print mean and stdev for chlorophyll overlying good bathymetric range locations.
```

3. Save and run the program.

PROBLEM 4 – There are eight daily SeaWiFS images in the directory: **/Users/YourUsername/data/tutorial_data/8day_files/** Write a python script called ***problem4.py*** to create an 8-day weekly composite-average from these eight files.

The end product of your program should be a single image that depicts the 8-day average chlorophyll conditions. Display the final average image.

See the *python tutorial handout* covering the **elegant method of composite averaging,** and section on ***glob.glob*** the section on ***imaging 2D data to the monitor***. Below are some basic hints to get you started:

- You need to import ***numpy, matplotlib.pyplot, matplotlib.colors*** and ***glob*** modules.
- Missing values in the SeaWiFS images are designated by ***NAN***
- Use ***glob.glob*** to get a *List* of all the eight file names

---

PROBLEM 5 – The directory **/Users/YourUsername/data/tutorial_data/wkly_swf1998/** contains a yearly time series of individual weekly average images of chlorophyll concentration in the Gulf of Maine region (**Lat: 37 to 46, Lon: -72 to -63**) *for all of 1998*. The imagery has the same pixel resolution and geographic bounds as in the previous Gulf of Maine imagery used in problems 3 and 4.

1. For each of the weekly average images, determine the **spatial average** chlorophyll concentration *within a small sub-region* defined by: **Longitude from –69 to –66***;* **Latitude from 38 to 40** (i.*e., a single number for each week that represents the sub-region's average chlorophyll value for that week*).
2. As you LOOP through successive weeks, assign the result of each spatial average value separately for each to a pre-specified "*holding vector*" that has as many elements are files in the input directory. Initialize the *holding vector* outside the FOR LOOP.
3. Finally, make an x-y (i.e., time-chl) graph (***see: plot function in the matplotlib.pyplot module***) depicting a graph of the week-by-week seasonal rise and fall of chlorophyll values in that small region through the year.

PROBLEM 6 – Suppose you went on a Gulf of Maine oceanography cruise in 1998 that went from **May 10 to June 27** and you wanted to find out what the average chlorophyll concentration was in the Gulf of Maine region for that time period. Using the daily chlorophyll data found in **/Users/YourUsername/data/tutorial_data/dly_swf1998/** derive a regional **image** of the average chlorophyll concentration from the daily images for the cruise time period.

Hint: You will need to:

- Import ***numpy, glob, os, matplotlib.colors*** and ***matplotlib.pyplot***
- Use **os.path.basename** to split the path from the root part of the fine name
- Extract the *Julian Day* out of the root (base) file names (*and convert ascii to integer*)
- Inside the cumulative loop, add an **if statement for jday >= min_min and jday <= max_day** and then inside this if statement you would open a file and add it to preexiting only for the correct files. This will loop through every single file in the directory, but will only open a file and use it for averaging if the file date within the proper date range.
- Finally average the chlorophyll for all those files to form a single average chlorophyll image (*as in problem 4*).
- Display the result.

PROBLEM 7 – There may be times when you will have an excel file containing data collected at sea that is in the form of columns containing, *Station_Date, Station_Time, Station_Latitude, Station_Longitude, Station_Measurement_Variable*. You can retrieve the satellite data for the corresponding station location contained in the text file by using a combination of tasks that have already been done in the previous problems (e.g., problem 1) - and the addition of one new task: reading an excel txt file using the python function readlines().

The text file ***fish_track_data.txt*** that is located in the directory ***/Users/YourUsername/data/*** contains the ***year, Julian day, hour, location*** (latitude and longitude) and ***fish tag number*** (Tag #100 and #101). The data columns are tab delimited.

Note: *In this very simple class example the tag data is for a single day, but in the real world this could have been for many days.*

1. Use *Sublime* (*or Excel*) to open the ***fish_track_data.txt*** file and just see what is inside the text file (*do not alter the file*). Then close the file.
2. Write a python program to read all of the *fish_track_data.txt* file information into python and then use its latitude/longitude information to navigate to the appropriate location in the satellite image that corresponds to the day and year and read in the satellite chlorophyll concentration at each fish observation point. The full file name of this "Gulf of Maine" satellite file (**37N to 46N and -72W to -63W**) you will use is:

   /Users/YourUsername/data/tutorial_data/mon_swf2006/A20060602006090_mar_chlor_a_f999x999.avg

3. I want you to then print the original station information contained in the ***fish_track_data.txt*** along with the new satellite information, to **a new text file**.

4. The following are some key points to keep in mind:
   - ❖ You will need to import numpy (use the form: ***import numpy as np***)
   - ❖ See the tutorial for reading in numerical binary data to read in the satellite file
   - ❖ See tutorial to read in all the ascii (text) file line (use the form: ***f1.readlines()***)
   - ❖ Once all lines are read in you will have a List of lines with each element of the List being one row (one full line) in the text file.
   - ❖ The first line in the header line with the column names. The data comes with the subsequent lines
   - ❖ Each of the line has a tab character (**'\t'**) separate each column and ends with a new line (**'\n'**). The '\n' need to be removed.
   - ❖ The following sections of the overall code should help a lot…

```
# read in the tab delimited text files of station information..
f1= open(fish_fname,'r')
all_lines= f1.readlines()
f1.close()
header_line= all_lines[0].strip()  # first line is the header (also remove '\n')
all_lines=all_lines[1:]                #remove header line from the List of lines

#open a new file for text output…
f2= open(new_fish_fname,'w')
f2.write(header_line + '\tchl\n')

for i in range(len(all_lines)):
    one_line= all_lines[i].strip()
    one_split_line= one_line.split('\t')
    year= one_split_line[0]
    jday= one_split_line[1]
    hr= one_split_line[2]
    lat= float(one_split_line[3])
    lon= float(one_split_line[4])
    fish_id= one_split_line[5]

    # still need to convert lat/lon to row/column
    # still need to read in 'station_chl' values at row/col location of satellite data

    f2.write(one_line + '\t' + str(station_chl) + '\n')

f2.close()
```

5. Save and run the python program

6. Open the output file with Excel

PROBLEM 8: There are 16 years of global annual average chlorophyll in the following directory: /Users/YourUsername/data/annual_global_chl.

1. Open a Terminal Window and cd to the directory containing the chlorophyll files.
2. Launch python and import my_hdf_cdf_utilities and use the hdf/cdf functions that were described in the python tutorial to do the following:

   - ❖ Find the name of the product contained in these files.
   - ❖ Find out if a slope/intercept needs to be applied to this data

3. Launch a new *Sublime* text document to make a python program that will create an annual global chlorophyll anomaly image for each year relative to the 16 year average conditions. Here are some hints…

   - ❖ The dimensions of the data vector are ydim=2160, xdim=4320
   - ❖ Since this is global data, the lat/lon dimensions are:

     south= -90.

     north= 90.

     west= -180.

     east= 180.
   - ❖ Since the files in this directory are netCDF files (rather than the binary data files you have been reading previously) you will need to read in the data using one of the functions I have created for you in my_hdf_cdf_utilities using this line of code:

     year_chl_img=read_hdf_prod(filename,'chlor_a')

   - ❖ Be sure to convert Fill_Values to NANs right after reading in your data.
   - ❖ Compute the 16-year composite average *"climatology"* using the approach of problem 4.
   - ❖ Next, Loop through each year (again) and compute the anomaly expressed as a percent difference relative to the climatology conditions using the following basic formula:

     anomaly_image= 100.0*(year_chl_img – climatology_chl_img)/climatology_chl_img

   - ❖ When you display the result using imshow(), make sure you are NOT taking the log10() of your anomaly data, and since you are scaling the anomalies to be a percentage of the 16-year mean, you should use vmin= -50.0 and vmax= 50.0
   - ❖ When looping through to create png images of the anomalies, please note:
     - o Use the **plt.close()** command after the **plt.savefig** command. This will clear out the graphic memory after each png and make thing clean for the next png in the next loop.
     - o So *not* use **plt.imshow()** and instead view your mapped images as pngs using the Desktop Finder.

- ❖ Save the image to a data directory of your own choosing. Note that you will need to make the output directory if it does not already exist (google: *python os.mkdir* to see how to make a directory from within your python script).
- ❖ Consider using the os.path.basename function in the os package to get the root names of each annual data file, and include those roots in the name of the corresponding anomaly output image
- ❖ Use the Mac Preview application to look at the output images.

Problem 9 – Mapping Level-2 Swath Data With PYRESAMPLE

There is a section of python code in the larger ocean color data processing scripts I have written that specifically maps Level-2 swath data. I have pasted this small mapping section of code into a small python module called:

problem_9_swath_map_utilities.py

For this problem 9 you will read in Level-2 Swath data and map the data to a Cylindrical coordinate map projection.

1. The python mapping utility (*problem_9_swath_map_utilities.py*) is located ~/***python_programs/utilities*** directory on your own computer.
2. Use the following Level-2 Swath file: *~/data/tutorial_data/A2010241173000.L2_LAC_OC.x.hdf*
3. Read use the ***chlorophyll***, ***latitude*** and ***longitude*** data from the hdf file.
4. Use the function called: ***map_l2_swath*** (*found in problem_9_swath_map_utilities.py*) to map the Level-2 data.

    - ❖ Note, you will need to open the file *problem_9_swath_map_utilities.py* with Sublime to see what argument input (and in what order) ***map_l2_swath*** needs to map the swath data.

5. **Note:** Use the following map boundaries and satellite resolution.
    - ❖ map_coords= [37.0, -72.0, 46.0, -63.0] #south, west, north, east
    - ❖ l2_resolution= 1000.0 # 1000 meter = 1km resolution
6. Open a New text document called ***problem_9_ swath_map.py*** and save it in your ***python_exercises*** directory.
    - ❖ You will need these modules imported into ***swath_map_problem_9.py***

```
import numpy as np
from my_hdf_cdf_utilities import *
from problem_9_swath_map_utilities import *
import matplotlib.pyplot as plt
import matplotlib.colors
```

7. Display an image of the swath data and the mapped data using matplotlib…
8. Look through the code found in **problem_9_swath_map_utilities.py** Maybe also have a look at: https://pyresample.readthedocs.io/en/latest/

The **OceanColor Web** (http://oceancolor.gsfc.nasa.gov) serves as a data source for Level-1, Level-2 and Level-3 products, but it is also an important source of information on data quality and a very important source of information on using SeaDAS.

In the next few days we will begin using SeaDAS software to process Level-1A (raw top-of-the-atmosphere) data to Level-3 (mapped geophysical data – e.g., chlorophyll concentration). This current exercise is intended to show you how to order Level-1, 2 and 3 data and this will then allow you to have data that covers your favorite part of the ocean when you start doing the SeaDAS processing exercises in the coming days.

1. ***Organize Your Data Folders*** **--** To be well organized it is best if your data are stored under sub directories that are created within a main data directory ***(~/data***).

   As you download new data sets during this exercise, they should all go in new subdirectories created within/under your current ***~/data*** directory

2. **NASA GESDISC DATA ARCHIVE Registration:**

1. Go To: **http://oceancolor.gsfc.nasa.gov**
2. Click On (*near the top of the page*): **SERVICES > Registration**
3. Click On: **Create an Account via Earthdata Login**
4. Fill in The Requested Information
5. Fill in the Requested Information
Please remember your EOSDIS username and password for later use.
6. Go to your user profile
7. https://urs.earthdata.nasa.gov/profile
8. Use the pulldown menu "My Applications"
9. Click on Authorized Apps
10. Select: "Approve More Applications"
11. Choose and Approve **NASA GESDISC DATA ARCHIVE**
12. Note go to the top of the webpage and use the "Edit Profile" Pulldown Menu on the main login page and then select the check boxes for the User Agreement for **MERIS** and separately for **Sentinel** data.
13. Click **Save Profile**
14. Now, separately, Open (*or create*) a **~/.netrc** file using Text Wrangler and add the following line:

machine ***urs.earthdata.nasa.gov*** login ***your_user_name*** password ***your_password***

…Where user name and password are the EOSDIS username and password from above registration

A. ORDERING *LEVEL-1A* MODIS-AQUA DATA

1. Launch your favorite web browser and Go to the OceanColor Web: http://oceancolor.gsfc.nasa.gov/

2. Click on the ***DATA*** then under ***Data Browsers*** click ***Level 1&2 Browser***

3. Notice that by default the MODIS (*Aqua*) product is selected (*small box is checked*) near the top left of the page

4. Click on ***your favorite month/year*** (*for now pick a summer month to improve chances of finding relatively cloud-free scenes*).

5. Select the bounding coordinates for the ocean region of your own interest, on the right side of the page, and then click [**Find Swaths**] -- *Try to limit the region (for right now) to no larger than 10 degrees of latitude and 10 degrees of longitude*.

> ***NOTE:*** *Normally (outside of this current demonstration exercise) at this point you would just click on the [Order Data] button at the top of the page and order all of the scenes in the month and this would include a lot of completely clouded over scenes. When we learn later in the class to run automated batch processing scripts it is no problem to let the computer to process right through the cloudy scenes. But for now, we will be more selective and select some of the best (most cloud-free scenes).*

NOTE --->Be sure to write down the latitude longitude bounds you use to order the data for later use when you process and mapping the data <--NOTE

6. To keep the data transfer (for right now) to a minimum, you will download about 10 or 15 files. Find a few files that are relatively cloud-free based on the small thumbnail of the scene. If you wish to see the image in a bigger size, click on the image. Select the image you like for ordering by ***clicking the series of dots just above the thumbnail image***. This will change the dots to the word '***Yes'***. Go to the next set of images by clicking on the small numbers to the left of the global map locator in the middle of the page.

7. When you have selected 10 or 15 good scenes, click the [**Order Data**] button located near the top right of the webpage.

8. Write your **EOSDIS *Username*** in the box at the top of the order page **<----- NOTE**

9. Select **[Do] extract** the data, on the left side of the page. **<----- NOTE**

10. On the right side of the page, re-enter your bounding latitudes and longitudes, around the image of your area of interest.
11. Check Level-1

12. Leave the last three boxes at the bottom of the page checked

13. Click [**Review Order**]

14. On the next page click [**Submit Order]**.

15. In just **one or two minutes** you should receive an email from dataorders@seawifs.gsfc.nasa.gov (*note: If you do not get an email within 5 minutes you should re-order your data*). The email is intended to make sure that a real human has confirmed that they want the data, so they can then begin staging the data for subsequent pick up/download. You have two ways to make the confirmation.

    a. You can simply hit the email *reply button* – which is how I normally do things.

    b. Alternatively, you can Click On the confirmation link within the email (you then may have to enter your SeaWiFS Authorized User Name and Password). In the webpage that pops up you then Click On the confirm button. *The confirm button may not be on the link immediately, but make sure to check back and click confirm once it does appear*.

    NOTE: Using the reply to the email to confirm your order does not always seem to work for some students in the class (I have no idea why). If you have chosen to use the email reply and you do not get a second email within 15-20 minutes you should re-order your data.

16. After replying to the first email to start the staging of your data, you will receive a second email (about 10 or 20 minutes depending on the data order size) saying that the data is beginning to be staged. **You should wait until you receive a third email** will come when ***staging is complete*** and you are then able to pick up your data via ftp. **Occasionally, these emails are picked up in your email SPAM folder, so be sure to check here if it's not in your inbox.** The email will have instructions on where to ftp the data from (i.e., what computer and what directory on that computer).

B. TRANSFERRING THE LEVEL 1 & 2 DATA FROM THE NASA COMPUTER USING *WGET*

1. Open a Terminal Window and change directories to ***~/data*** (type: ***cd ~/data***).
2. Now make a new subdirectory called *aqua_l1a* (type: ***mkdir aqua_l1a)***.
3. Now ***cd*** into the new ***aqua_l1a*** directory.
4. Starting from within your *aqua_l1a* directory
   (type ***pwd*** to be sure you are in the right directory – is should be:
   (***~/data/aqua_l1a …or equivalently … /User/Your_NetID/data/aqua_l1a***).
5. Open the email that was sent to you that said staging of your data was complete. Note the ***manifest file url*** in the message. **Copy this full url!!**

6. Type the following in the open Terminal window.

obdaac_download –v –-http_manifest '*manifest file url*'

where, *manifest file url* is **the url that you copied from your email in step 5 above** regarding your data being staged and ready for transfer. You will now paste it into the line of code in the terminal window.

For Example:

obdaac_download -v --http_manifest 'https://oceandata.sci.gsfc.nasa.gov/ manifest.txt?h=ocdist303'

NOTE: Keep the quotation marks around the full manifest url.

7. You should see signs in the terminal window that downloading is proceeding.
8. When the Unix prompt returns the data transfer is complete
9. Type: ***ls*** to confirm that the files have been transferred to your directory.

10. **NOTE:** If you ordered/requested a large number (ca., greater than 20) files, they will be delivered in a series of tar files (e.g., requested_files_1.tar, requested_files_2…). All tar files together contain your full ordered data.

You can either *cd* to the data directory containing your tar files and one-by-one type the following Unix command: ***tar –xvf requested_files_1.tar***

or else… Use a small python script contained in my_general_utilities by typing

python
from my_general_utilities import *
untar_ocweb(directory_path_where_tar_files_reside)
exit()

C. ORDERING *LEVEL-1A* SEAWIFS DATA

Follow the same procedure above, with the following important differences, to order SeaWiFS Level-1A data

1. Check the **SeaWiFS MLAC** box for data product type in the upper left of the webpage under the SeaWiFS heading - and uncheck the aqua box

2. Click on the "**Reconfigure Page**" button

3. Select August 2004 or earlier years (Unless US Coast, then any year)

4. Write in your latitude and longitude bounds and then click on <***Find Swaths***>

5. After you have selected 5-10 data files that you are interested in, click Order Data. Then on the next page, Check Level-1

6. Everything else is the same as above except make a new directory on your computer called ~/data/***seawifs_l1a*** and then *cd* into this new directory before starting the ***wget*** data transfer process with the same commands described above.

D. ORDERING *LEVEL-2* MODIS-AQUA DATA

Follow the same procedure as above, with the following important differences.

1. Check **Level 2 OC** data (not Level-1). By just checking **Level 2**, you will receive all the available level-2 products, not including SST (chlor_a, normalized water-leaving radiances, aerosol, etc.)

2. If you wish to receive only specific level-2 products then, check the following:
   Check **Level 2 OC,** and **chlorophyll a** to receive only chlor_a data
   Check **Level 2 OC,** and **chlorophyll a**, and **Level 2 SST (11 um)** to receive chlor_a and sea surface temperature data only.

3. Make new directory called ***~/data/aqua_l2*** and when you receive the third email stating that your data is Staged, begin the data transfer using the *wget* method.

E. ORDERING *LEVEL-2* SEAWIFS DATA

Follow the same procedure as above with the following important differences.

1. Check **SeaWiFS MLAC** box and uncheck **MODIS** on the first page

2. Check **Level 2 OC** data (not **Level-1**). By just checking Level 2, you will receive all the available level-2 products of SeaWiFS (chl_a, normalized water-leaving radiances, aerosol, etc.)

3. If you wish to receive only specific level-2 products then, check the following: Check **Level 2**, and **chlorophyll a** to receive only chl_a data

4. Make new directory called ~/data/seawifs_l2 to download your data via ftp.

F. ORDERING VIIRS LEVEL-1 AND LEVEL-2 DATA

1. Following exactly the methods for ordering MODIS/SeaWiFS data except check the VIIRS box and then click on the **Reconfigure Page** button before selecting your region of interest.

VIIRS Level-1A Data

When making small orders of VIIRS level 1 data might get just a list of files that begin with ***https://SOME_PATH…./Vyyyydddhhmmss…*** To download these files can either do one file at a time with the wget command or run a small python script that I wrote to automatically go through an entire list of files.

1. Create a new data directory where you want to store the data that you will be retrieving (e.g., *~/data/viirs_l1a*)
2. Then *cd* into that new data directory.

If doing one file at a time, simply type:

swget ***https://SOME_PATH…./Vyyyydddhhmmss…***

If downloading lots of files, use my python script with the following steps:

3. Open a new text file in your newly created data directory using Sublime.
4. **COPY *the lines*** that each begin with http:// from the full list and **PASTE** them into to the open Sublime text document and then **SAVE** the text document. Be sure that only lines of text that start with http:// are in this saved text file (i.e., no header lines of text). NOTE: *You can save this text file with any name of your own choosing (e.g., my_file_list.txt)*
5. Then run the *wget_oceans_gsfc* procedure by typing the following at the python prompt:
   from my_general_utilities import wget_oceans_gsfc
   wget_oceans_gsfc*(fname*)

   where fname is the name of the text file containing the list of satellite files names that you created above (e.g., wget_oceans_gsfc('my_file_list.txt').
6. When the programs ends, exit python by typing *exit()* and then type *ls* to see the files in your data directory.

G. ORDERING SEAHAWK DATA

1. Following exactly the methods for ordering MODIS/SeaWiFS data except check the SeaHawk box and then click on the **Reconfigure Page** button before selecting your region of interest.

H. ORDERING MERIS FRS LEVEL-2 DATA (OPTIONAL)

1. Following exactly the methods for ordering SeaWiFS data except check the MERIS box and then click on the **Reconfigure Page** button before subsetting.

2. NOTE: Keep the latitude and longitude boundaries small (e.g ***delta lat/lon of around 3 or 4-degree***). MERIS data is 350m resolution and so large lat/lon bounds can create very large file!

I. ORDERING AND RETREIVING MERIS LEVEL-1 DATA (OPTIONAL)

NOTE: this is an ***OPTIONAL*** exercise that you should only do if you are sure that you want to work with MERIS Level-1 data – Many of you will be fine with MERIS Level-2 data) and so this retrieval method that is specific to Level-1 data will not be necessary.

1. Following exactly the methods for ordering SeaWiFS data except check the MERIS RR or FRS box and then click on the **Reconfigure Page** button before give the lat/lon coordinates.
2. NOTE: Keep the latitude and longitude boundaries small (e.g ***delta lat/lon of around 3 or 4-degree***). MERIS FRS data is 350m resolution and so large lat/lon bounds can create very large file!
3. Select **[Do] extract** the data, on the left side of the page. **<----- NOTE**
4. On the right side of the page, re-enter your bounding latitudes and longitudes, around the image of your area of interest.
5. MERIS Level-1 data requires EOSDIS Registration which you should have completed at the start of this data ordering handout.

   a. **For multiple files --**

   Start with the email that tells you the data has been staged and paste the following from your email into the terminal window:

   ```
   obdaac_download -v --http_manifest 'manifest url'
   ```

   Now modify this line by replacing ***manifest-url*** with ***your own url*** from the email sent to you regarding your data being staged and ready to be transferred.

   b. **For a single file --**

   swget https://url_path/basename

   See this page for more details:
   http://oceancolor.gsfc.nasa.gov/forum/oceancolor/topic_show.pl?tid=3081

   c. **If you run into problems, try this…**

   ```
   touch ~/.urs_cookies

   curl -b ~/.urs_cookies -c ~/.urs_cookies -L -n -O http://your_full_URL_filename
   ```

J. ORDERING AND RETREIVING HICO LEVEL-1 DATA (OPTIONAL)

Retrieving HICO Level-1 data (no Level -2 data available at this time) follows the same basic steps used to order and retrieve MERIS Level-1 data. This also includes the need to use your EOSDIS username and password in the *swget* command to retrieve your data.

K. Searching and Retrieving Sentinel-2 Data:

1. Go To the following link and fill in the requested information
2. https://scihub.copernicus.eu/dhus/#/self-registration
3. Respond the confirmation email

4. Go To: https://scihub.copernicus.eu/dhus/#/home
5. Click on the Tab to the left of the "Insert Search Criteria" bar to expand the search GUI
6. Check the small box for Mission: Sentinel-**2**
7. Select the Following
   a. Satellite Platform: (S2A or S2B)
   b. Product Type: **S2MSl1C (Level 1 data)** or try S2MSl2A (Level 2 data *not tested here*)
   c. Make your Favorite Box for the ROI (keep it small) by holding down the right mouse button (or Control Key + holding down regular mouse button).
   d. Click back on the "Insert Search Criteria" bar and hit the Return key.
8. Select the best file based on location and cloud cover
9. Go to the Cart and wait awhile (maybe even a day!) for the product to go from Offline to Online and ready to be downloaded.
10. When the file is no longer Offline, then click on the download icon.

L. Searching and Retrieving *Level-3 Data* from the Ocean Color Web Portal

I have written a python procedure that can help with the retrieval of large amounts of Level-3 data from the oceancolor website. The python procedure is called ***wget_oceans_gsfc.py*** and is located at: *~/python_programs/utilities*/my_general_utilities.py

If you view this procedure using a text editor, you will find some instructions on how to search for the data you want and then retrieve the data to your own computer.

To demonstrate the steps needed to search and retrieve data, you should try to find and retrieve monthly or 8 day or daily Level-3 Standard Mapped Images (SMI) of Aqua Chlorophyll at 4km resolution. You can pick any month or year from January 2003 to present.

7. Go to: ***http://oceandata.sci.gsfc.nasa.gov*** and click on the *"Customizable File Search"* link
8. Specify the data type (pattern) In this example choose… ***MO*CHL_chlor_a*4km*** *(the wild card asterisk symbols are needed).*
   Note that ***not a single white space can lead or follow the search string****.*
9. Select Data Type (in this example choose: *Level3 SMI*)
10. Select the mission (in this example choose: MODIS *Aqua*)
11. Specify a date range
12. Check the first "***three"*** boxes near the bottom of the page (*leave the checksum box un-checked*).
13. Click the Search Button
    The search will find the names of all the MODIS-Aqua chlorophyll files at 4-km

resolution. A header line of text describing the number of files found and list of file names will be displayed in the web page (or as a text file) that can then be used by wget_oceans_gsfc.pro to retrieve the listed files.

14. Create a new data directory where you want to store the data that you will be retrieving (e.g., *~/data/aqua_SMI*)
15. Then *cd* into that new data directory.
16. Open a new text file in your newly created data directory using Sublime.
17. **COPY *4 or5* *lines*** that each begin with https:// from the full list of file names (4 or 5 only *for this small class example*) found in the webpage of the search results and **PASTE** them into to the open Sublime text document and then **SAVE** the text document. Be sure that only lines of text that start with http:// are in this saved text file (i.e., no header lines of text). NOTE: *You can save this text file with any name of your own choosing (e.g., my_file_list.txt)*

To retrieve your files and store them into your current data directory…

18. *cd* to the data directory described above…
19. To run the *wget_oceans_gsfc* procedure by typing the following at the python prompt:
    from my_general_utilities import wget_oceans_gsfc
    wget_oceans_gsfc*(fname)*

    where fname is the name of the text file containing the list of satellite files names that you created above (e.g., wget_oceans_gsfc('my_file_list.txt').

20. When the programs end, exit python by typing *exit()* and then type *ls* to see the files in your data directory.

INTRODUCTION TO INTERACTIVE SEADAS PROCESSING PART 1: SEAWIFS PROCESSING LEVEL-1 TO MAPPED LEVEL-3

Motivation – The purpose of this lab is to demonstrate in a graphical manner the basic steps involved in processing SeaWiFS ocean color satellite data from raw multispectral radiance values to a mapped image depicting the regional distribution of chlorophyll. The processing begins with a Level-1A file containing top-of-the atmosphere radiance values recorded by the satellite's onboard radiometer. The first step is Level-2 processing which takes the 'top-of-the-atmosphere' radiance intensities in the Level-1A file and performs atmospheric corrections to derive a Level-2 file of normalized water leaving radiances (L_{nw}), chlorophyll concentration, a few other geophysical parameters and quality control flags. A second step (Level-3 processing) takes the geophysical data contained in the Level-2 file and maps it from the raw satellite perspective to a user-specified coordinate system (oftentimes a cylindrical system is chosen where each pixel is of equal degrees of latitude and longitude).

Launch SeaDAS:

Just double-click on the SeaDAS icon that is located in your Toolbar at the Bottom of the screen (or on your Desktop).

NOTE: If any of the processors loaded onto the computer are corrupt, go to the **OCSSW** menu at the top, and click **Update OC Processors**. Check the box of the corrupt processor, then click **Apply** and **Run**.

A. DISPLAY A RAW SEAWIFS L1A FILE:

1. To load a file into SeaDAS, use **Open Product** located under **File > Open…**

2. Locate S2007240173520.L1A_MLAC.x.hdf in: ~/**data/interactive_seawifs** and select **[Open Product]**.

   The file content will appear in the ***File Manager Pane*** to the left. Double click on the filename to see the **Bands** directory.

3. View the available bands for display by expanding the ***Bands*** folder found in the File Manger pane. To display the band as an image, *double-click* the band. **Begin by displaying the 412 nm band** (L1A_412).

   You can resize the window for better image viewing using the **Zoom** (slider bar) and the **Pan** (left/right/up/down Arrows) tool located in the upper left side of the open image viewer panel.

4. Add a coastline using **Layers > Coastline, Land and Water** function found in the **Icon Tool Menu** at the top to the window (*it looks a map icon of Italy*).

5. Repeat step 3 twice more using 510 nm and 865 nm respectively. Note the increasing contrast between water and land at longer wavelengths. Remember that all images are

a grey scale of intensity of just a single narrow wavelength of light. This is NOT like a black and white photo that incorporates the intensity of a broad range of wavelengths!

6. Close all open image displays by clicking the **[X]** in the top left corner of the viewing window.

7. Close the file by *right-clicking* on the filename in the File Manager Window.

B. PROCESS A LEVEL-1A FILE TO A LEVEL-2 (ESTIMATED GEOPHYSICAL VARIABLES SUCH AS NORMALIZED WATER LEAVING RADIANCES AND CHLOROPHYLL CONCENTRATION)

1. Using the Menu Bar at the very top of the computer screen, choose **SeaDAS-OCSSW** > **l2gen…** to bring up the SeaDAS SeaWiFS L2 File Generating Program GUI.

   *Make sure the **Main** tab is highlighted at the very top left side of the GUI.*

2. In the **Primary I/O files Section** – Click on the […] button to select the file you viewed earlier **(~/data/interactive_seawifs/ S2007240173520.L1A_MLAC.x.hdf**)

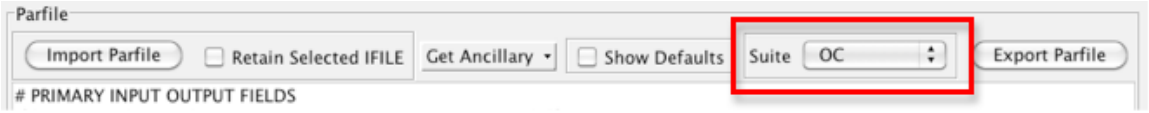
3. An output file name for the Level 2 output will automatically be selected for you. You may keep it or change it as you wish.

4. Under the **Parfile** section, click on the ***[Get Ancillary]*** button to get the necessary Meteorological (MET) and Ozone (OZONE) file needed for the atmospheric correction that takes place in the Level-1 to Level-2 processing.

5. Press the [**Products**] **Tab** (*next to the Main Tab at the top of the open window*) to get a list of the many products that l2gen can compute…

   The products to be generated are listed in the **Selected Products** *box located at the bottom of the display window*. You can select additional products by expanding any of the categories under **Product Selector** and checking the box next to the product name.

   *The default products for **l2gen** are set by the parameter suite. For the purpose of this example, we are using the default ocean color suite, **OC** that shows up under the Main Tab.*



6. All other options and settings under the additional tabs of the L2 File Generating Program will be automatically set to default. Run the processing by pressing **[Apply]** followed by **[Run]**.

7. Press OK when the processing is complete to close the l2gen GUI.

8. Remote the original Level-1 file from the File Manager window by right-clicking on it and closing all products.

C. DISPLAY THE LEVEL-2 CHLOROPHYLL IMAGE AND LOOK AT QUALITY CONTROL FLAGS.

1. Open the Level-2 file generated in the previous section using **File > Open Product**.
2. Expand the **Bands** folder to view all of the Level-2 products produced.
3. Display the **chlor_a** band by *double-clicking* on it in the **File Manager** window.
4. To vary color map applied to the image, go to the **Color Manipulation Pane** *located below the* **File Manager Pane.**

   Select a color **Palette** using the drop-down menu.

   Use the **Min** and **Max** boxes under **Range** to adjust the minimum and maximum values set to the color scheme (try min: 0.1 and see what happened).
5. To load quality control (QC) flags for display, open the **Layer Manager Pane** *located to the right of the viewing window*. If this pane is not visible, go to the menu at the top of the computer screen and select **Layer > Layer Manager**.
6. To load a mask, simply check the box located next to the left of the name. You can change the properties of any mask, such as color or transparency, simply by clicking on the property.
7. Load multiple masks and view the image in the display window. Next select only the **LOWLW** mask and view the image in the display window. The image will automatically refresh to display only the currently selected masks.
8. Close all open menus by clicking the **[X]** in the top left corner.
9. Close all open image displays by clicking the **[X]** in the top left corner.

Note that the Level-2 chlorophyll image you produced is still in satellite perspective with all sorts of spatial complication (e.g., mirror parallax earth curvature). The next processing step is to take this Level-2 image and map it to well defined cylindrical mapped coordinate system.

D. PROCESS LEVEL-2 DATA TO LEVEL-3

Level-2 to Level-3 processing entails mapping data from the raw satellite perspective to cylindrical coordinates. *The first step is to spatially bin the file (**l2bin**), followed by creating a temporal average for the period of the satellite observation (**l3bin**).*

1. From the menu bar, select **SeaDAS-OCSSW** > **l2bin…** to bring up the Level-2 Binning GUI.

2. Click **[…]** on the far right of the **infile** to bring up the Select Input File GUI. Select the **L2 file** (level-2 file) you created in step B3, and press **[Select].**

3. Specify the **Output file name,** which is the file name for the Level-2bin (spatial bin) file that will be created

   Note: ---> It is likely an output file name will automatically be generated for you that has *"**L3b**"* as part of its name. **You definitely should replace the *L3b* with *L2b* for**

the output file name. So, in the present case the l2b output file name should be: S2007240173520.***L2b***_Day_OC

4. Specifying **resolve** will set the resolution of the output file, with **H** representing .5 kilometers, and each number specifying the resolution in kilometers. The default resolution is 9km. **Select 4 to set the resolution to 4km.**

5. For **l3bprod**, specify **chlor_a** as your output products by typing chlor_a in the box.

6. For **prodtype** select **Day**.

7. For **oformat**, specify **netCDF4**

8. ***De-select*** the box labeled **Open in SeaDAS** so that the output file is *not* automatically opened as a product.

9. Click **[Apply]** and select [**Run**] and wait a few minutes for the processing to be complete (check the shell window to monitor the processing progress.)

   Note: Normally you would select all products for future application, but for this demonstration the chlor_a is sufficient.

10. From the menu bar, select **SeaDAS-OCSSW** > **l3bin…** to bring up the Level-3 Binning GUI.

11. Click **[…]** on the far right of the **infile** to bring up the Select Input File GUI. Find the file that was just created in using **l2bin** (S2007240173520.***L2b***_Day_OC), and press **[Select].** *(Note: if you did not change L3b to L2b in previous l2 bin step, then this file will probably have a name with L3b when in fact it is an L2b file generated with the l2bin step above – I suggest deleting this file and redoing the l2bin step above using L2b as part of the output name.)*

12. Specify the **Output file,** which is the file name for the Level-3 binned file that will be created. SeaDAS will automatically adjust the suffix of the **infile** to create the name of the **ofile** for you (*and it should have a L3b in it under normal operations*). So, in this case the **output file name should be: S2007240173520.*L3b*_Day_OC**

13. Specify latitude limits (**latsouth**, **latnorth**) as: **37** and **46**

14. Specify longitude limits (**lonwest**, **loneast**) as: **-72**, **-63**

15. For **oformat**, specify **netCDF4**

16. Un-Check the box labeled **Open in SeaDAS** so that the output will not try to automatically open as a product.

17. Click **[Apply]** and select [**Run**] and wait a few minutes for the processing to be complete (check the shell window to monitor the processing progress.)

E. CREATE A MAPPED IMAGE FROM THE LEVEL-3 BINNED FILE.

This step will create a mapped version of the l3bin data and write it would as a 2D floating point array of the geophysical data contained in a NetCDF4 or HDF file.

1. From the menu bar, **SeaDAS-OCSSW** > **l3mapgen…** to bring up the l3mapgen GUI.

2. Click ***[…]*** on the far right of the **ifile** to bring up the Select Input File GUI. **Find the Level-3 binned file that was just created in using l3bin (S2007240173520.*L3b*_Day_OC)**, and press **[Select].**

3. Just ignore the output file name right now, since SeaDAS will build a filename for you when you go through and specify parameters for the remaining fields in the window.

4. Specify the product to map in the **product** field. Simply enter **chlor_a**.

5. For **projection**, select **platecarree** in the drop down menu to specify a cylindrical projection.

6. Specify latitude limits (**latsouth**, **latnorth**) as: **37** and **46**.

7. Specify longitude limits (**lonwest**, **loneast**) as: **-72**, **-63**.

8. Specify a resolution by selecting **4km** in the drop down menu under **resolution**.

9. Under the **scale_type** drop down menu, select **linear**

10. De-Select **apply_pal**

11. ***Select*** the box labeled **Open in SeaDAS** so that the output file is automatically opened as a product.

12. Check to make sure the **ofile** name at the top of the window is entered as: **S2007240173520.*L3m*_DAY_chlor_a_4km.nc**

13. Leave all other fields as the default value

14. Click **[Apply]** and select [**Run**] and wait a few minutes for the processing to be complete (check the shell window to monitor the processing progress.

F. DISPLAY THE LEVEL-3 MAPPED IMAGE OF CHLOROPHYLL AND THE QUALITY CONTROL FLAGS.

1. If your file is not already located within the **Product View** window, use **Open Product** located under **File > Open Product** to locate and open your standard mapped image file. If a popup menu regarding netcdf file input standards just click ok.

2. Expand the **Rasters** folder. Highlight the mapped **chlor_a** and then double click to bring up the display window containing the mapped chlorophyll image.

3. If you want to change the color scale, go to the **Color Manipulation Pane** *located just below the File Manager Pane*. Select a color scheme in the drop down menu. You can change the **Min** and **Max** values in the display if you wish.

4. To end the SeaDAS program close all SeaDAS GUIs and **Quit SeaDAS**

INTRODUCTION TO INTERACTIVE SEADAS PROCESSING PART 2: MODIS PROCESSING LEVEL-1 TO MAPPED LEVEL-3 *(OPTIONAL)*

NOTE: MODIS processing involves more steps than SeaWiFS processing. Unlike SeaWiFS L1A files, MODIS L1A files cannot simply be combined with the corresponding MET and OZONE files and then processed to L2 files. The MODIS L1A files must first be combined with Attitude and Ephemeris files to create a GEO file. The GEO file is used along with the L1A file to create an L1B file. The L1B file can then be combined with the GEO file and corresponding MET, OZONE, and SST files to be processed to L2 and then L3 files in a manner identical to SeaWiFS L1A processing.

Launch SeaDAS: *Just double-click on the SeaDAS icon that is located on your desktop.*

A. GENERATE A MODIS-AQUA GEO FILE FROM L1A.

9. Using the menu bar, choose **SeaDAS-OCSSW** > **modis_GEO…** to bring up the *modis_GEO.py* Processor GUI.

10. In the **Primary I/O files** pane, Click On the ***[…]*** button, located on the far right, to bring up the *Select Input File* GUI. Select **~/data/interactive_modis/A2010241173000.L1A_LAC.x.hdf** and press **[Select].**

11. SeaDAS should automatically specify a name for the Output GEO file in the output field, or you can make your own. I suggest sticking with **A2010241173000.GEO.x.hdf** SeaDAS will also automatically locate the proper *Attitude* and *Ephemeris* files needed for the GEO file generation.

12. Hit ***[Run].*** The prompt will tell you when the file has been generated.

B. GENERATE A MODIS-AQUA L1B FILE.

1. Using the menu bar, choose **SeaDAS-OCSSW > modis_L1B …** to bring up the ***modis_L1B Processor*** GUI.

2. Specify the ***ifile*** by first clicking on ***[…]*** button located to the right. Select **A2010241173000.L1A_LAC.x.hdf**. Press [Select].

3. SeaDAS attempts to automatically specify the Input MODIS GEO file, If it fails to find to GEO file, then click on the ***[…]*** button located to the right. Select the GEO file you created in the previous step. And select the GEO file you generated in the previous step.

4. Hit [Run]. The prompt will tell you when the L1B file has been generated.

   Note: L1B_LC.x.hdf is the 1km resolution file, L1B_HKM.x.hdf is the 500m resolution special bands and L1B_QKM.x.hdf is the 250m resolution special bands.

C. PROCESS THE L1B MODIS-AQUA FILE TO A L2 FILE.

1. Using the menu bar, choose **SeaDAS-OCSSW > l2gen…** to bring up the l2gen window.

2. Make sure the ***Main*** tab is selected at the very top left side of the GUI.

3. In the **Primary I/O files** section, click on the ***[…]*** button located on the far right side of **ifile** to bring up the Select Input File GUI. Select the *Level-**1B** file* that you created (**A2010241173000.*L1B*_LAC.x.hdf**).

4. The geofile file should be automatically selected, but it not, then click on the ***[…]*** button located to the right. Select the GEO file you created in the first step. Press [Select].

5. The Level 2 output file will automatically be assigned, and you may keep it (recommended) or change it as you wish.

6. Under the **Parfile** section, the section titled Click on the ***[Get Ancillary]*** button to get the necessary Meteorological (MET) and Ozone (OZONE) file needed to best atmospheric correction that takes place in the Level-1 to Level-2 processing.

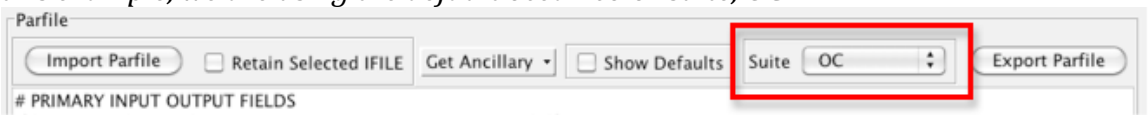
7. Press the [**Products**] tab to get a list of the many products that l2gen can compute. The products to be generated are listed in the **Selected Products** box at the bottom of the display window. You can select additional products by expanding any of the categories under **Product Selector** and checking the box next to the product name.

8. All other options and settings under the additional tabs of the L2 File Generating Program will be automatically set to default.

9. Be sure to uncheck the box marked **open in SeaDAS** at the bottom.

10. Pressing **[Apply]** followed by **[Run]**.

11. *NOTE: The default products for **l2gen** are set by the parameter Suite. For the purpose of this example, we are using the default ocean color suite, **OC**.*



12. Press OK when the processing is complete to close the l2gen GUI.

D. PROCESS MODIS AQUA LEVEL-2 DATA TO LEVEL-3

Once a Level-2 file is made using SeaDAS the steps to go from Level-2 to L2bin to L3bin and finally to L3_Map **are identical no matter the satellite sensor.**

Processing the modis-aqua Level-2 file you just generated to Level-3 mapped is optional. If you want to generate the mapped modis-aqua file, please use steps D, E and F above that were used to process SeaWiFS from Level-2 to Level-3 and just substitute in the modis-aqua Level-2 file.

SEADAS PROCESSING: PYTHON COMMAND MODE

Motivation -- After processing a single image using the interactive mode of the SeaDAS system, you probably now realize its limitation with regard to processing large numbers of images. However, using SeaDAS in command mode circumvents this limitation and makes SeaDAS a very powerful satellite data processing software for managing large satellite data sets. However, this power does not come without some cost to the user in the form of needing to learn python scripting language (or other computer scripting language if you want). Used in combination with python scripts, SeaDAS processing software can automate general satellite data processing tasks applied to large data sets. With just a few key strokes you can set the processing scripts working through the night processing Level-1 files to Level-2, mapping the Level-2 files to a defined coordinate system and then creating daily or weekly or monthly composite images. It removes a lot of the drudgery of remote sensing and gives the oceanographer more time to apply the observations to oceanographic questions – which to me is the best part of satellite oceanography. I have written some python scripts that make it easy to change and organize the data processing output. We will be working with these processing scripts for the next couple of days but before we start using these scripts, it will be instructive to first run a much simpler example script that will allow you to clearly see the steps involved with processing satellite data with SeaDAS functions contained within a python processing script. This simple script contains all of the basic SeaDAS commands that are used in the more elaborate scripts. After running this simple example of a processing script, you will then be introduced to a relatively more elaborate set of scripts that have more "bells and whistles" thrown in to make processing output more organized (at the cost of making the scripts a bit more difficult to see clearly what is going on).

PROBLEM 1: READING AND THEN RUNNING A SIMPLE PYTHON PROCESSING SCRIPT…

I have created a simple python processing script called ***class_demo_Batch_Proc.py*** located under ***~/python_programs/tutorial_programs***. This relatively simple script contains the basic SeaDAS commands for processing MODIS-Aqua Lelve-1A files to Level-3 mapped images of surface chlorophyll concentration.

Your job is to first open this simple script with a text editor (e.g. *Sublime*) and read down through the script to see how the filenames of the MODIS Aqua Level-1A data are read into a python *List* and how this list is of filenames is then used within a simple ***for loop*** to processes each file sequentially through each basic step until a mapped file is finally

created. Then you can run the demo python script and wait for the processing to complete to see your newly created mapped files. This might take several minutes for each file. There are three example Level-1A files to be processed in this exercise.
NOTE.

1. The Leve-1A data input directory in ***class_demo_Batch_Proc.py*** needs to be changed at *line 19 (open class_demo_Batch_Proc.py with Sublime*).

   indir = ~/data/aqua_l1a_small_batch

2. The Level-3 output directories in ***class_demo_Batch_Proc.py*** needs to be changed at lines 28 and 32.

   odir= ~/data/aqua_l1a_small_batch_output
   pngdir= ~/data/aqua_l1a_small_batch_output/png

3. After the processing is complete, navigate to the Level-3 directory to view the PNG output files located in the output directory.

PROBLEM 2: PROCESSING LEVEL-1 DATA TO LEVEL-3

To process your own **Level-1A Data** (*that you downloaded from the OceanColor Web the other day*) you will need to run the suite of python programs that I have already created. These python scripts have very similar SeaDAS commands that you worked with in the simple class demo script used in problem #1. The big difference is all the book keeping going on automatically determine which satellite sensor you are working with and what kind of final product you want to produce and where you want the final product to be stored on the computer.

1. Using *Sublime*, Open the following file: ***~/python_programs/Batch_Proc.py*** and **Read the entire file carefully before running the program!**

   NOTE: The sole purpose of this *front-end* script is to setup the processing conditions that will then be passed to the ***batch_L12.py*** (processing from Level-1 to Level-2) and ***batch_L23.py*** (processing Level-2 to Level-3).

   It is these last two scripts that actually do all the hard processing work!

2. To set up the ***Batch_Proc.py*** script so that it will process **YOUR** Level-1 data to Level-3, you need to modify the ***Batch_Proc.py script*** in the following way...

   a. Change the satellite data levels (Level-1, Level-2 or Level-3) for the starting and stopping level. In your case now, you want to process from Level-1 (starting level *"**1**"*) to Level-3 (ending level *"**3**"*).

   b. ***Change the Lat/Lon limits*** to suit your own geographic region.
   The coordinate order must be: [***south, west, north, east***] <--------<<<

   c. Change the **input directory** path (***l1dir***) to the path of your own L1A data directory.

d. Change the **output directory** path for your Level-2 and Level-3 output directories (i.e., for ***l2dir and mapdir***) to a new Level-3 output directory of your own choosing:

For Example…

if you have a Level-1 directory path of…
/Users/your_user_name/data/aqua_l1a …

Then a reasonable Level-2 directory path would be…
/Users/your_user_name/data/aqua_L1-L2output

and a reasonable Level-3 directory path would be…
/Users/your_user_name/data/aqua_L1-L3output

3. Set the temporal averaging to ***DLY*** to output a daily mapped file for each of daily Level-1A files in your input directory.

4. ***Save*** the changes you made above and then run your newly modified batch processing file using the following steps:

   a. Launch A Unix Terminal (unless you already have one open)
   b. Making sure you have ***cd*** to in your ***~/python_programs*** directory
   c. Type: ***python Batch_Proc.py***

PROBLEM 3: PROCESSING LEVEL-2 DATA TO LEVEL-3

Process the **Level-2 data** that you downloaded the other day to Level-3 in a manner very similar to the previous problem, except that you will use Level-2 data as your starting point.

1. Open your ***Batch_Proc.py*** file with a text editor.

2. Change the satellite data levels for the starting and stopping level. Now you will set, the starting level for Level-2 (starting level ***"2"***) and to final process will again be Level-3 (ending level ***"3"***).
3. ***Change the Lat/Lon limits*** to suit your own geographic region. The coordinate order is [***south, west, north, east***]

   NOTE: The ordering of the lat/lon bounds MUST BE: south, west, north and then east

4. Change the **input directory** path (***l2dir***) to the path of your own Level-2 data directory.
5. Change the **output directory** path for your Level-2 and Level-3 output directories (i.e., for ***l2dir and mapdir***) to a new Level-3 output directory of your own choosing:

   For Example:
   Then a reasonable Level-2 directory path would be…
   /Users/your_user_name/data/aqua_L2

   and a reasonable Level-3 directory path would be…
   /Users/your_user_name/data/aqua_L2-L3output

6. *Save* the changes you made above and then run your newly modified batch processing file by:

    a. Use a Unix Terminal Window and make sure you are in the ***~/python_programs*** directory, then:

    b. Type***: python Batch_Proc.py***

7. Check out the mapped png results

PROBLEM 4: CHANGING THE TEMPORAL AVERAGING PERIOD FOR LEVEL-3 OUTPUT

Referring back to problem *2* above where you created Level-3 Daily (DLY) data from Level-1 data. Suppose now you want to create a *monthly* composite for the same data? You might think that you need to go back and reprocess your Level-1 data back to Level-3 as you did in problem 2 using the new time average period set to monthly (MON), **but you would be wrong**!

The time consuming set in problems *2* was generating the Level-2 files from Level-1 data and once the Level-2 data is made, these files should always be used to generate new kinds of temporal averaging. Averaging Level-2 data into *Weekly* or *Monthly* composite averages is very fast compared to reprocessing from Level-1 to Level-2.

So, go back to the Level-2 data directory that was used for output in **problem 2** and use this for Level-2 to Level-3 process in the same manner you did with problem 3.

NOTE: Use the same Level-3 output directory that was used in problem 2. The python scripts will automatically make new sub-directories called: monthly for your output under the l3dir. This will put your monthly data in the same main output directory (*l3dir*) as your original daily output.

Think of the *l3dir* as the main directory for output and the python scripts then automatically generate subdirectories based on product type (e.g., chlor_a) and temporal averaging.

PROBLEM 5: PROCESSING DATA FROM OTHER SENSORS

Repeat Problems 2, 3 and 4 using: *SeaWiFS, VIIRS, HICO, MERIS, SEAHAWK and/or OLCI data*

PROBLEM 6: PROCESSING DATA FROM PACE OCI

The PACE Mission will not launch for another year, but simulated PACE data has been created for a hypothetical month/year

I have downloaded a single Level-1B file and three Level-2 files containing three different groups of level-2 data products. All for the same hypothetical day and location (Gulf of Maine).

~/data/pace-oci/L1/PACE_OCI_SIM.20220321T165718.L1B.V9.1.nc

~/data/pace-oci/L2-AOP/PACE_OCI_SIM.20220321T165500.L2.OC_AOP.V8.nc
~/data/pace-oci/L2-IOP/PACE_OCI_SIM.20220321T165500.L2.OC_IOP.V8.nc
~/data/pace-oci/L2-BGC/PACE_OCI_SIM.20220321T165500.L2.OC_BGC.V8.nc

AOP files contain apparent optical properties (eg. Rrr_nnn)
IOP files contain inherent optical properties (e.g., adsorption and scattering)
BGC files contain derived biological parmeters (e.g., chlor_a, CDOM)

a) Use the BatchProc.py script to process the PACE L1 file to L3 containing chlor_a and Rrd_nnn.

b) Use the BatchProc.py script to process the PACE L2 AOP to L3 containing Rrs_nnn.

c) Use the BatchProc.py script to process the PACE L2 BGC to L3 containing chlor_a.

d) Cd into each of the L2 directories and use the ncdump -h function to see what is inside these netCDF files.

NOTE: The Simulated L1B PACE data is found here:
https://oceandata.sci.gsfc.nasa.gov/directdataaccess/Level-1B/PACE-OCI/2022/080/

NOTE: The Simulated L2 PACE data is found here:
https://oceandata.sci.gsfc.nasa.gov/directdataaccess/Level-2/PACE-OCI/2022/080/

PROBLEM 7: READ THROUGH THE PYTHON CODE FOR L1 TO L2 AND L2 TO L3 PROCESSING

Open batch_L12.py and batch_L23.py with a text editor and make your best attempt to follow the coding logic used to make these programs work. The Level-1 to Level-2 and Level-2 to Level-3 programs are found in the directory called: **~/python_programs**.

You will be using these scripts a lot after you leave the course and if you read them now and have me around to ask questions, things will be much clearer to you later on when I am not close by...

PROBLEM 8: ALTER LIST OF LEVEL-2 PRODUCTS GENERATED IN LEVEL-1 TO LEVEL-2 PROCESSING

Using the Level-1 data that you started with in Problem 2, reprocess the data to Level-2 and then Level-3, BUT this time change the Level-2 products produced to a new set of products. For example, you could try outputting different chlorophyll pigment algorithms or add in a cdom product to the output list.

To add new products, open the Batch_Proc.py main processing front plate and scroll down to the section called: "*Optional L1 -> L2 Processing Variables*"

The look for the variable named: ***prod_list_L2***

Add a few new products to the list (separated by comma). For example:

Prod_list_L2= 'chlor_a, chl_gsm,cdom_index'

Note: To get a list of the valid products that l2gen can produce there are several options. Starting with the best option first:

1. USING THE SEADAS COMMAND CALLED: *GET_PRODUCT_INFO*

From within a Terminal Window type***: get_product_info*** to get the following help…

Usage: **get_product_info [option=val] <productName>**

Options:

-help (boolean) (alias=-h,--help) (default=false) = print usage information

-version (boolean) (alias=--version) (default=false) = print the version information

-dump_options (boolean) (alias=--dump_options) (default=false) = print information about each option

-dump_options_paramfile (ofile) (alias=--dump_options_paramfile) = print information about each option to paramfile

-dump_options_xmlfile (ofile) (alias=--dump_options_xmlfile) = print information about each option to XML file

-l (boolean) (default=false) = list all of the products

-r (boolean) (default=false) = list all of the products recursing through the wavelengths of the sensor specified sensor (string) (default=MODISA) = sensor name (or ID) to use for wavelength expansion or product lookup

<productName> product to print detailed information about

For Example, Type: ***get_product_info -l sensor=SeaWiFS***

A SPECIAL NOTE ABOUT THE MAPPING METHOD USED BY BATCH_L23.PY

The ***batch_L23.py*** script deals with the mapping problem that has the option to use two different approaches depending on your preference or the spatial resolution of the Leve-2 data that will be mapped.

If the spatial resolution of the Level-2 file is 1 km
then the standard approach is used to go from Level-2 to Level-3:

> Level-2 file ***> l2bin > l3bin > l3mapgen*** to generate mapped data file > then ***mapplotlib-cartopy*** for continent masking and coastline generation for the png files

If the spatial resolution of the Level-2 file is < 1 km (*e.g., modis hires or meris frs or hico*) --- *or if you set straight_map= 'yes' in the BatchProc.py script* --- then the following approach is used to go from Level-2 to Level-3

Level-2 file ***> resample.py*** to generate mapped data file (using nearest neighbor approach) > then mapplotlib-cartopy for continent masking and coastline generation for the png files. (*see: https://pyresample.readthedocs.org/en/latest/)*

This approach does not impose the same spatial lower limit on the mapped output and so is a good compromise to achieve mapped images of higher spatial resolution that matches the higher spatial resolution of these special data products.

VARYING THE QUALITY CONTROL FLAGS (L2_FLAGS) IN THE BATCH PROCESSING SCRIPTS

The level of quality control applied to a satellite image is determined by the numbers of l2_flags (quality control flags) that are examined. There are 32 possible flags to examine. If a given l2_flag is set to be examined, then wherever (*pixel by pixel*) the quality level for that particular flag exceeds predefined limits (*e.g., mirror angle exceeded a certain number of degrees or atmospheric aerosol model undetermined*), then QC processing will flag that pixel by filling it with a *NAN*. The predefined limits that trigger the qcflag can be changed, but this is somewhat beyond the scope of this course.

Setting the batch processing to check for *fewer* l2_flags, means marginal quality data will not be flagged as bad. This yields greater data retention in the image, but some of the data pixels may have questionable quality. By setting the processing to check *more* l2_flags, you are increasing the confidence of the data by throwing out marginal quality data – you lose a lot of data in the image, but the data you do have are more likely to reflect reality.

To change the number of flags to be checked:

1. Open Batch_Proc.py in a text editor
2. Scroll down the file to find the line containing: ***color_flags=***
3. You have the following choices for this variable:
    a. ***color_flags=****'standard'*
    …this forces the processing script to go to the seadas installation and open the default flags parameter file and use the standard flags used by the Goddard group in producing their global product processing

b. ***color_flags=****'comma separated list*' of the flag names that you want to apply to the final level-3 file output *(see full list below*).
…for example: ***color_flags=*** *'ATMFAIL, LAND, CLDICE, HIGLINT' would only apply these flags to remove suspect data before writing out the Level-3 file*

FULL LIST OF LEVEL-2 QUALITY CONTROL FLAGS (BOLD == STANDARD FLAGS USED BY GODDARD)

Bit	Name	Short Description	L2 Mask Default	L3 Mask Default
00	ATMFAIL	Atmospheric correction failure		ON
01	LAND	Pixel is over land	ON	ON
02	PRODWARN	One or more product algorithms generated a warning		
03	HIGLINT	Sunglint: reflectance exceeds threshold		ON
04	HILT	Observed radiance very high or saturated	ON	ON
05	HISATZEN	Sensor view zenith angle exceeds threshold		ON
06	COASTZ	Pixel is in shallow water		
07	spare			
08	STRAYLIGHT	Probable stray light contamination	ON	ON
09	CLDICE	Probable cloud or ice contamination	ON	ON
10	COCCOLITH	Coccolithophores detected		ON
11	TURBIDW	Turbid water detected		
12	HISOLZEN	Solar zenith exceeds threshold		ON
13	spare			
14	LOWLW	Very low water-leaving radiance		ON
15	CHLFAIL	Chlorophyll algorithm failure		ON
16	NAVWARN	Navigation quality is suspect		ON
17	ABSAER	Absorbing Aerosols determined (disabled?)		ON
18	spare			
19	MAXAERITER	Maximum iterations reached for NIR iteration		ON
20	MODGLINT	Moderate sun glint contamination		
21	CHLWARN	Chlorophyll out-of-bounds		
22	ATMWARN	Atmospheric correction is suspect		ON
23	spare			
24	SEAICE	Probable sea ice contamination		
25	NAVFAIL	Navigation failure		ON
26	FILTER	Pixel rejected by user-defined filter OR Insufficient data for smoothing filter ?		
27	spare	(used only for SST)		
28	spare	(used only for SST)		
29	HIPOL	High degree of polarization determined		
30	PRODFAIL	Failure in any product		
31	spare			

The list of default flags is found in $OCSSWROOT/share/sensor_name

Example: SeaWiFS default flags are located in the file:

$OCSSWROOT/share/seawifs/l2bin_defaults.par

SST DATA: WHERE TO GET IT AND HOW TO READ IT INTO PYTHON

The example sst data that you will be downloading should go in an sst directory that is under your personal data directory that is already on ~/data. Make a new sst directory under your personal data directory by typing:

mkdir ~/data

MODIS AQUA/TERRA SST (4KM-LEVEL-3-GLOBAL) 2001-PRESENT

If your SST data needs span 2001 to present, the MODIS-Aqua and MODIS-Terra is a very good data choice. If you need to go back further in time, then AVHRR Pathfinder or GHRSST is a good choice.

Selecting the MODIS SST Global Level -3 Data…

Referring back to the Ocean Color Web ordering and retrieving of Level-3 MODIS….

1. Go to: https://oceancolor.gsfc.nasa.gov
2. ***Click On: Data > Data File Search***
3. Specify the data type (pattern) In this example choose… ***DAY*NSST*4km***
   - *DAY*NSST*4km* --> *daily images of night time SST (NSST) at 4km resolution.*
   - ***Note*** *that* ***not a single white space*** *can lead or follow the search string.*
4. Select Data Type (in this example choose: *Level3 SMI*)
5. Select the mission (in this example choose: MODIS *Aqua*)
6. Specify a date range
7. Check the first "***three"*** boxes near the bottom of the page (*leave the checksum box un-checked*).
8. Click the Search Button
9. **Create a new data directory** where you want to store the data that you will be retrieving (e.g., ***~/data/sst/aqua_SMI***)
10. Then *cd* into that new data directory.
11. Open a new text file in your newly created data directory using Sublime.
12. **COPY *4 or5 lines*** that each begin with http:// from the full list of file names (4 or 5 only *for this small class example*) found in the webpage of the search results and **PASTE** them into to the open Sublime text document and then **SAVE** the text document. Be sure that only lines of text that start with http:// are in this saved text file (i.e., no header lines of text). NOTE: *You can save this text file with any name of your own choosing (e.g., my_file_list.txt)*

Retrieving Selected MODIS Level-3 Data…

1. *cd* to the data directory described above…
2. To run the *wget_oceans_gsfc* procedure by typing the following at the python prompt:
   from my_general_utilities import wget_oceans_gsfc
   wget_oceans_gsfc*(fname*)

where fname is the name of the text file containing the list of satellite files names that you created above (e.g., wget_oceans_gsfc('my_file_list.txt').

3. When the programs end, exit python by typing *exit()* and then type *ls* to see the files in your data directory.

READING AND IMAGING MODIS GLOBAL SST INTO PYTHON…

Global Level-3 MODIS data is in netCDF4 format. To read these files into python you will use the following three python programs.

hdf_prod_info.py …to get product name/spelling of the SST product
hdf_prod_scale.py …to get slope/intercept or fill value info needed to be applied to the DN
read_hdf_prod.py …to read the sst data – which is in DN format (not geophysical)

Step 1: From the command line in the terminal window, use the ***hdf_prod_info*** function and the ***hdf_prod_scale*** function to query the product name and any masking / scaling that might need to be applied to the stored sst data:

1. ***cd*** into the data directory containing your aqua sst data
2. Launch python
3. Type: *from **my_hdf_cdf_utilities** import **
4. Type: *import numpy as np*
5. Type: ***hdf_prod_info('thename_of_your_sst_file')***
   to find the list of products contained in the netCDF4 file (e.g., sst)
6. Type: ***hdf_prod_scale('thename_of_your_sst_file', 'sst')***
   to determine if a slope/intercept (scale_factor, add_offset) that needs to be applied to the data. The output of this function will also indicate whether there is a fill value (bad data) that should be set to NaN. In the present sst case it is seen that the slope=0.005 and intercept= 0.0 If slope had been 1.0 and intercept had been 0.0, the stored data is not DN, but rather in geophysical units. For my file, the fill value is -32767 and should be set to NaN.

Step 2: Now write out the following code into a new python script (*text file*) to read in the data sst and image it.

```
from my_hdf_cdf_utilities import *
import numpy as np
import matplotlib.pyplot as plt

#Read in unscaled SST data
dn_sst_array=read_hdf_prod(sst_fname, 'sst')
print(dn_sst_array.shape)
```

```
# Use mask and scale info from hdf_prod_scale to convert to SST
# Where "scale" "intercept" and "fill_value" are the numerical values determined
# using hdf_prod_scale function.  Note: this will convert DN values to geophysical
# values (degree C)
bad_loc=np.where(dn_sst_array==fill_value)
sst=dn_sst_array*scale + intercept
sst[bad_loc]=np.nan   #NaN out the fill values after scaling to avoid type conversion
                      #error (integer to float)
#Read in quality file associated with SST data:
qual=read_hdf_prod(sst_fname,'qual_sst')

#NaN out the bad quality data
# missing data qual_sst = -1, poor quality sst data  = qual_sst = 3-poor or 4-fail.
bad_qual=np.where((qual == -1) | (qual > 2))
sst[bad_qual[0],bad_qual[1]]=np.nan

#Image the SST data
mycmap=plt.get_cmap('nipy_spectral').copy()
mycmap.set_bad('k')
plt.imshow(sst,cmap=mycmap,vmin=-2,vmax=35)
plt.show()
```

PATHFINDER DATA 5.3 **1981-Present**

The Pathfinder SST product is a long-term data set of SST derived over the years from a long series of AVHRR sensors. All sensors have been carefully cross-calibrated to ensure a consistent time series of SST. This data is 4km resolution.

Retrieve Pathfinder SST Data

1. Create a new directory: mkdir ~/pathfinder_sst
2. Go To: https://www.ncei.noaa.gov/products/avhrr-pathfinder-sst
3. Click On: HTTPS
4. Click On: Your favorite year
5. Click On: data
6. Click On: any night time data file
7. Move: the downloaded file to ~/pathfinder_sst

Read-In Pathfinder SST Data to Python and Display

A. Using the following code to read in and quality mask Pathfinder SST data.

```
import matplotlib.pyplot as plt
import numpy as np
from my_general_utilities import *
from my_hdf_cdf_utilities import *

ifile='the sst file you downloaded in the previous step'

prod= 'sea_surface_temperature'
scale_factor= 0.01
add_offset_k = 273.15  # for units of Kelvin
add_offset_c = 0.0  # for units of Celsius
missing= -32768

qual_name='pathfinder_quality_level'
# quality_level:     0b, 1b, 2b, 3b, 4b, 5b
# quality_meanings:  no_data, bad_data, worst_quality, low_quality, acceptable_quality,
#                    best_quality

sst= read_hdf_prod(ifile, prod).squeeze()
qual=read_hdf_prod(ifile, qual_name).squeeze()

print('\n\nsst shape', sst.shape, '\n\n')

missing_data= np.where(sst == missing)
low_quality=  np.where(sst < 4)

sst= sst*scale_factor + add_offset_c
sst[missing_data]= np.nan
sst[low_quality]= np.nan
```

B. Display Pathfinder SST data.

See section on displaying satellite images to the monitor. Use linear colorbar. Use vmin= -2.0 and vmax=35.0 for best results.

C. Downloading a Lot of Pathfinder Data

I have written a program that will download a month of daily pathfinder sst date with a single call. In theory, you could place this call in a python loop that you could write and download as much data as you want (a month of daily data for each loop). For for now, just grab one month as practice.

The following commands will download daily data for January 1997.

1. Open a terminal window and change directory to **~/data/sst**
2. Make a new subdirectory within ~/data/sst called **avhrr_pathfinder** and then cd into that new subdirectory.
3. Launch python and type the following commands:
4. from my_general_utilities import *
5. **wget_avhrr_pathfinder_sst**('1997', '01') # the arguments are ***year*** and ***month***. Note the month must be **2** characters. *You are welcome to try a different year (between 1981 and present) and different month.*

GROUP FOR HIGH RESOLUTION SST (GHRSST) DATA (2002-PRESENT)

The GHRSST project produces high-resolution SST products derived from many individual sensors blended together. For more details, see: https://www.ghrsst.org/ghrsst/

See these Products Choices: https://www.ghrsst.org/ghrsst-data-services/services/

RETRIEVE L3 GHRSST GRIDDED DATA….

NOTE: GHRSST L3 Gridded products are useful for applications that require gridded SST at the highest possible resolution, as the L3 SST values have not been "smoothed" by spatial or temporal interpolation to fill in data gaps. Applications include coastal weather forecasting and coastal marine biological studies such as forecasting coral bleaching.

To Retrieve L3 Gridded Data:

1. Create a directory to store the sst data you will be downloading for example **~/data/sst/ghrsst/**
2. Go to: https://www.ncei.noaa.gov/access/ghrsst-long-term-stewardship-and-reanalysis-facility/
3. Click on: HTTPS
4. Click on: L4 > GLOB > JPL > MUR
5. Click on: **Favorite Year > Julian Day**

IMAGING GHRSST SST DATA …

I have made a small read function that will read in sst data and mask out the bad quality flags. Type the following code into a new python script to use my function to read in the data and image it. ***Note:*** *This is a very big file and will take a minute or two to load and display.*

```
from my_general_read_utilities import *
import matplotlib.pyplot as plt

sst= read_ghrsst_sst(fname)
print(sst.shape)

mycmap = plt.get_cmap('nipy_spectral').copy()
mycmap.set_bad('k')
plt.imshow(flipud(sst), cmap=mycmap, vmin=-2, vmax=35)
plt.show()
```

PYTHON SCRIPT FOR RETRIEVING LARGE AMOUNT OF GRHSST DATA

I made a small function for getting large amounts of GRHSST SST data.

The function is called ***wget_grhsst_sst*** and it is located in:
~/python_programs/utilities/my_general_utilities.py.

Take the following steps use this program:

1. Create a data directory on your computer where you want to store the data stored.
2. Change to that directory
3. Launch python
4. Type (*for example*) the following to get a few ghrsst sst data files for 2018:

```
from my_general_utilities import  *
wget_ghrsst_sst(year=2018, start_day=1,  end_day=2)
```

QUIKSCAT VECTOR WINDS FROM REMOTE SENSING SYSTEMS:

1. Create a folder under your home data directory, where you can save the ssmi files.
   a. ***cd*** *~/data*
   b. ***mkdir*** *quikscat*
2. Visit https://remss.com for your wind data
   Note: You must be a registered user (see: https://remss.com/register/)
3. Go To: The ***Missions*** pulldown menu (upper left of page)
4. Click On: **QuikScat**
5. Click On: **HTTP Button**
6. Click On: **bmaps_v##**
7. Pick you **favorite year > favorite month.**
8. Pick a Daily File (this will be any file **with a shorter file name**).
   Avoid the 3day average files (d3d.gz)

READING QUIKSCAT DATA INTO PYTHON

Wind Problem 1:

Uncompress your quikscat file if it is still compressed. Note: the file is still compressed if the name ends with .gz. Uncompress by typing:

gunzip the_name_of_your_file.gz

Write a python script to read in your quikscat data using the **read_quikscat_wind** function in **my_general_read_utilities**.

NOTE: *The read wind program returns an array that is 1440x720x2 where:*

wind_uv[0,:,:] is the global image of u (zonal) wind velocity
wind_uv[1,:,:] is the global image of v (meridional) wind velocity

positive u is wind directed eastward
positive v is wind directed northward

units are: meters/sec

missing data have to be quality masked for rain contamination using NANs

As an exercise do the following tasks:

1. Image the u (zonal) and v (meridional) components of wind speed.
2. Compute and image the magnitude of the wind speed $(u^2 + v^2)^{1/2}$ and display the result.
3. Draw the wind velocity vector field on top of the wind speed magnitude plot using the matplotlib.pyplot quiver function. The quikscat wind vectors are stored upside down, so you will need to use the flipud() function when imaging. **note:** you will need to change the vmin, and vmax values to go from 0 to 20 (or something close to this) to get the best colors.
4. Compute the vector wind ***stress*** components from the u and of v wind ***speed*** values using the formula:

   stress= ρ_{air}*C_D*U^2 **(**units are N/m^2)

   Where,
   ρ_{air} is the density of air = 1.2 Kg/m3
   U^2 is the wind velocity squared for either the u or v components
   C_D is the drag coefficient and can be formulated a lot of different way. Here are two you might consider. Pick either one or else if you want you gave give both a try…

   $C_D = 1.2 \times 10^{-3}$ *(Trenberth 1989) or…*
   C_D = (0.29 + 3.1/speed + 7.7/speed^2)/1000.0 *(Yelland and Taylor 1996)*

Display your results as per the wind speed data above in step.

NOTE>> *when displaying the wind stress components, you may try the values vmin=-1.0 and vmax=1. Compute and display the magnitude of the wind stress (u_stress2 + v_stress2)1/2 and display the result.*

Use the following code to get you started:

```
#!/usr/bin/env python

from my_general_read_utilities import *
import matplotlib.pyplot as plt
import numpy as np
from my_general_utilities import *

#Read quikscat vector wind into 3D array with
#dimensions (wind_components, xdim, ydim)
wind_uv= read_quikscat_wind('~/data/quikscat/quikscat_file')

#Separate wind components into 2D arrays
u_component = wind_uv[0,:,:]
v_component = wind_uv[1,:,:]

#Note: In normal practice these dimensions can be found using:  print u_component.shape
xdim=1440
ydim=720
```

```
mycmap = plt.get_cmap('nipy_spectral').copy()
mycmap.set_bad('k')

#Display u component of wind speed (do the same for v component)
plt.figure(1)
plt.imshow(flipud(u_component),cmap=mycmap,vmin=-10.0,vmax=10.0)

#Compute and display wind speed
wind_speed = np.sqrt(np.square(u_component)+np.square(v_component))
plt.figure('wind speed magnitude')
plt.imshow(flipud(wind_speed),cmap=mycmap,vmin=0.0,vmax=20.0)

#Display vector winds arrows
#Reduce size of u and v components to provide data for arrow field
smxdim=80
smydim=40
u=rebin_down_nan(flipud(u_component), smydim, smxdim)
v=rebin_down_nan(flipud(v_component), smydim, smxdim)

Y=np.mgrid[0:40]*(ydim)/(smydim-1)
X=np.mgrid[0:80]*(xdim)/(smxdim-1)

plot1 = plt.figure('wind speed magnitude')
plt.quiver(X, Y, u, v,          # data
      color='w',                # color the arrows based on this array
      cmap=cm.seismic,          # colour map
      headlength=5)             # length of the arrows

plt.show()            # display the plot
```

PYTHON SCRIPT FOR RETRIEVING LARGE AMOUNTS OF QUIKSCAT WIND DATA (*OPTIONAL*)

I made a small function for getting large amounts of QuikScat Wind data (a full year at a time).

The function is called ***wget_quikscat*** and it is located in:
~/python_programs/utilities/my_general_utilities.py.

Take the following steps use this program:

5. Create a data directory on your computer where you want to store the wind data
6. Change to that directory
7. Launch python
8. Type (*for example*) the following to get a a full year of quikscat wind data for 2007:

```
from my_general_utilities import  *
wget_quikscat('2007')
```

9. **Note:** QuikScat was operational from **1999 to 2009**

SSM/I SCALAR WIND SPEED:

RETRIEVING SSMI DATA…

1. Create a folder under your home data directory, where you can save the ssmi files.
   a. ***cd*** *~/data*
   b. ***mkdir*** *ssmi*
2. Visit https://remss.com for your wind data
   Note: You must be a registered user (see: https://remss.com/register/)
3. Go To: The ***Missions*** pulldown menu (upper left of page)
4. Click On: **SSMI**
5. Click On: **HTTP Button** (left list)
6. Pick your favorite spacecraft (eg. **f14 or f15 or f16**)
7. Pick you **favorite year > favorite month.**
8. Pick a Daily File (this be any file **with a shorter file name**).
   Avoid the 3day average files (d3d.g)

READING SSMI DATA INTO PYTHON…

Note: Each daily ssmi file contains a global **morning (am)** and a global **evening (pm)** image **in the same file**. The read function I have written will average the am/pm images to get a single daily average. Data are converted to geophysical values by the read program. Any bad or missing data are set to NAN.

The ***read_ssmi_speed*** function is found in ***my_general_read_utilities***

Use the following steps to read in an ssmi wind speed file:

1. *Uncompress* your SSMI file if it is still compressed by typing the following at a Unix Terminal Window:

gunzip the_name_of_your_file.gz

2. The **read_ssmi_speed** function returns a single 2-D array of scalar wind speed by using the following step:
   a. Launch python from a unix terminal window
   b. type the following lines at the python prompt

```
from my_general_read_utilities import *
wind_speed_array= read_ssmi_speed('/path_to_your_data/the_name_of_your_file')
```

3. Display the 2-D global wind data using the basic python display steps for simple 2-D array of data. Note: Speed is not a vector and does not have a u_component and v_component. It just had a magnitude – a single value at each pixel location. *Try using vmin=0.0 and vmax=25.0 for the data display limits.*

Retrieving large amounts of SSMI data (*Optional*)

I made a small function for getting large amounts of SSMI Wind data (a full year at a time.

The function is called ***wget_ssmi*** and it is located in:
~/python_programs/utilities/my_general_utilities.py.

Take the following steps use this program:

10. Create a data directory on your computer where you want to store the wind data
11. Change to that directory
12. Launch python
13. Type (*for example*) the following to get a a full year of ssmi wind data for 2015:

    from my_general_utilities import *
    wget_ssmi('2015',*'the satellite you want – see note below'*)
    for example: wget_ssmi('2015', 'f18')

14. **Note:** SSSMI was operational from **1987 to Present**
    But the sensor has flown on successive satellites:
    satellite **== f08(1987-) sequentially to f18(2010-present)**

ASCAT Vector Winds:

The first Advanced Scatterometer (ASCAT) was launched on the EUMETSAT MetOp-A satellite in October 2006. It became fully operational in May 2007 and continues to operate today. Another ASCAT instrument became operational on MetOp-B when launched in September 2012. Both satellites, MetOp-A and MetOp-B carry identical ASCAT instruments. The main objective of ASCAT is the measurement of wind speed and direction over the oceans, though ASCAT is also used for studying polar ice, soil moisture and vegetation.
http://www.remss.com/missions/ascat/

Retrieving ASCAT Data…

1. Create a folder under your home data directory, where you can save the ascat files.
   a. ***cd*** *~/data*
   b. ***mkdir*** *ascat_files*

2. Go to: **http://www.remss.com/**

3. *You now need to register as a data user. See:* http://register.remss.com
4. Click on **Measurements > Wind** Pulldown Menu
5. Scroll Down to and Click On: **ASCAT**
6. Click on the **HTTP Button** (left list)
7. Pick your favorite spacecraft (eg. **metopa or metopb or metopc**)
8. Click On: **bmaps_v02.1**
9. Pick you **favorite year > favorite month.**
10. Pick a Daily File (this be any file **with a shorter file name**).
    Avoid the 3day average files (d3d.gz)

READING IN ASCAT DATA TO A PYTHON PROGRAM…

Each daily ascat file contains a global **morning (am)** and a global **evening (pm)** image **in the same file**. The read function I have written will average the am/pm images to get a single daily average. Data are converted to geophysical values by the read program. Any bad or missing data are set to NAN.

Since ascat measures **vector** winds, the read function returns a 3-D set of data containing an global image of the u_component of wind and a global image of the v of wind.

The ***read_ascat_wind*** function is found in ***my_general_read_utilities***

Use the following steps to read in an ascat vector wind file:

1. *Uncompress* your ascat file if it is still compressed by typing the following at a Unix Terminal Window:

   gunzip the_name_of_your_file.gz

2. The **read_ascat_wind** function returns a single 3-D array of vector wind speed by using the following step:
   a. Launch python from a unix terminal window
   b. type the following lines at the python prompt

   ```
   from my_general_read_utilities import *
   uv_vector_wind_array= read_ascat_wind('/path_to_your_data/the_name_of_your_file')
   u_component= uv_vector_wind_array[0, :, :]
   v_component= uv_vector_wind_array[1, :, :]
   ```

3. Display the u_component and the v of the vector winds using the basic python display steps used for for the quikscat vector wind data above. *Try using vmin=-10.0 and vmax=+10.0 for the data display limits.*

Retrieving large amounts of ASCAT data (*Optional*)

I made a small function for getting large amounts of ASCAT Wind data (a full year at a time.

The function is called ***wget_ascat*** and it is located in:
~/python_programs/utilities/my_general_utilities.py.

Take the following steps use this program:

15. Create a data directory on your computer where you want to store the wind data
16. Change to that directory
17. Launch python
18. Type (*for example*) the following to get a a full year of ascat wind data for 2007:

    from my_general_utilities import *

    wget_ascat('2015',*'the satellite you want – see note below'*)
    for example: wget_ascat('2017', '**`metopsa`**')

19. **Note:** ASCAT was operational from **2016 to Present**
    But the sensor has flown on successive satellites:
    satellite == **metopsa**(2016-present), **metopsb**(2012-present), **metopsc**(2019-present)

WINDSAT VECTOR WINDS:

The WindSat Polarimetric Radiometer was developed by the Naval Research Laboratory (NRL) Remote Sensing Division and the Naval Center for Space Technology for the U.S. Navy and the National Polar-orbiting Operational Environmental Satellite System (NPOESS) Integrated Program Office (IPO). It was launched on **January 6, 2003** aboard the Department of Defense **Coriolis Satellite**. WindSat was meant to ***demonstrate*** the capabilities of a fully polarimetric radiometer to measure the ocean surface wind vector from space. Prior to launch, the only instrument capable of measuring ocean wind vectors were scatterometers (active microwave sensors). In addition to wind speed and direction, the instrument can also measure sea surface temperature, soil moisture, ice and snow characteristics, water vapor, cloud liquid water, and rain rate. http://www.remss.com/missions/windsat/

RETRIEVING WINDSAT DATA…

1. Create a folder under your home data directory, where you can save the ascat files.
   a. ***cd*** *~/data*
   b. ***mkdir*** *ascat_files*
2. Visit https://remss.com for your wind data
   Note: You must be a registered user (see: https://remss.com/register/)

3. Go To: The ***Missions*** pulldown menu (upper left of page)
4. Click On: **WindSat**
5. Click On: **HTTP Button** (left list)
6. Click On: **bmaps...**
7. Click Through: **your favorite year** > **your favorite month**
8. Pick a Daily File (this be any file **with a shorter file name**).
   Avoid the 3day average files (d3d.gz)

READING IN WINDSAT DATA TO A PYTHON PROGRAM…

Each daily WindSat file contains a global **morning (am)** and a global **evening (pm)** image **in the same file**. The read function I have written will average the am/pm images to get a single daily average. Data are converted to geophysical values by the read program. Any bad or missing data are set to NAN.

Since WindSat measures **vector** winds, the read function returns a 3-D set of data containing an global image of the u_component of wind and a global image of the v of wind.

Note: windsat produces 3 different types of wind products.

WSPD_LF – Low Frequency Wind Speed (uses low microwave wavelengths)

WSPD_MF - Low Frequency Wind (uses medium microwave wavelengths)

WSPD_AW – All Weather Wind Speed

The read function I have written can read in any one of these wind products (see method below for selecting a given wind product). See the following for a description of the wind products: http://www.remss.com/missions/windsat/

The ***read_windsat_wind*** function is found in ***my_general_read_utilities***

Use the following steps to read in an windsat vector wind file:

1. *Uncompress* your windsat file if it is still compressed by typing the following at a Unix Terminal Window:

   gunzip the_name_of_your_file.gz

2. The **read_windsat_wind** function returns a single 3-D array of vector wind speed by using the following step:
   a. Launch python from a unix terminal window
   b. type the following lines at the python prompt

```
from my_general_read_utilities import *
windprod= 'WSPD_AW'        #Choices are: 'WSPD_LF', 'WSPD_MF' or 'WSPD_AW'
```

```
uv_vector_wind_array= read_windsat_wind('/path/the_name_of_your_file', windprod)
u_component= uv_vector_wind_array[0, :, :]
v_component= uv_vector_wind_array[1, :, :]
```

3. Display the u_component and the v of the vector winds using the basic python display steps used for for the quikscat vector wind data above. *Try using vmin=-10.0 and vmax=+10.0 for the data display limits.*

RETRIEVING LARGE AMOUNTS OF WINDSAT DATA (*OPTIONAL*)

I made a small function for getting large amounts of WindSat Wind data (a full year at a time.

The function is called ***wget_windsat*** and it is located in:
~/python_programs/utilities/my_general_utilities.py.

Take the following steps use this program:

20. Create a data directory on your computer where you want to store the wind data
21. Change to that directory
22. Launch python
23. Type (*for example*) the following to get a full year of windsat wind data for 2007:

```
from my_general_utilities import  *
wget_windsat('2007')
```

24. **Note:** WindSat was operational from **2003 to Present**

ALTIMETRY DATA FROM COPERNICUS MARINE ENVIRONMENT MONITORING SERVICE (CMEMS)

Becoming A Registered CMEMS Data User

1. To Register (Avoid Safari, Use Firefox or Chrome) Go To:
   http://marine.copernicus.eu/services-portfolio/register-now/

Getting Altimetry Data from CMEMS (After Successfully Registering as a User)

1. Go To:
   https://data.marine.copernicus.eu/product/SEALEVEL_GLO_PHY_L4_MY_008_047/description
2. Click On: [Data access]
3. Click On: MOTU Link for: cmems_obs-sl_glo_phy-ssh_my_allsat-l4-duacs-0.25deg_**P1D**
4. Click On: [Clear all]
5. Then select just:
   a. Sea surface height above geoid - *adt* [m]
   b. Surface geostrophic eastward sea water velocity - *ugos* [m/s]
   c. Surface geostrophic northward sea water velocity - *vgos* [m/s]

6. Set Lat/Lon -180 to 180 and 90 to -90
7. Select a 1-month range: e.g., Jan-1 to Jan-31.
8. Click On: [**Downloads**] – found at the top of the page.
9. Make a new directory calls ***altimetry*** under your ***~/data directory***
10. Move the Downloaded Data to ***~/data/altimetry/***
11. You are all done!

NOTE: The single netCDF altimetry file you downloaded will contain a series of daily data for the time-period you requested. In addition to the daily data, there is also a vector with the calendar dates that go with each daily image.

1. Write a python script to read in absolute dynamic height data using the read_ altimetry_adh function contained in my_general_read_utilities using the following lines of python code:

   ```
   from my_general_read_utilities import  *

   adh_dates, adh_data=read_altimetry_adh('the_name_of_your_altimetry_file')
   ```

   a. Now display just one of the days. So, use: single_adh= adh_data[0] and then display the single_adh data to the monitor.

   b. Include in the python script the commands needed to display the dynamic height file using vmin=0.0 and vmax=2.0 for your colorbar range.

   c. You will need to use the np.flipud() function to turn the map right-side-up when imaging.

2. Now write a second script that will read in the absolute geostrophic velocity. Start with the following lines of code:

   ```
   from my_general_read_utilities import  *

   uv_dates, u_data, v_data= read_altimetry_uv('the_name_of_your_altimetry_file')
   ```

   a. Display u and v for a single day (using two separate figure windows for each). So, use: single_u= u_data[0] and single_v=v_data[0] and then display data.

   b. Use values of vmin=-0.5 and vmax=0.5 for the color bar range.

   c. You will need to use the np.flipud() function to turn the map right-side-up when imaging.

EXAMPLE OF USING EOF ANALYSIS

I have written a python program called ***class_eof_svdc.py*** (*located in your python_programs directory*) that does an EOF analysis on global *monthly* SST data for the years 1995 through 2004. This period contains the large 1997-1998 El Nino event and this event shows up clearly in the analysis.

The program is also set up to do an EOF analysis on *annual* global chlorophyll. But you will need to open the python file with a text editor to change things from using sst to chl.

This program serves as an example of what an EOF can do **and as a python template for you to modify in the future** if/when you decided to do an EOF analysis on your own data set.

1. cd into *python_programs* directory

First… Use *Sublime* to open the python script: **class_eof_svdc.py**

On Line #33
Change the directory path to point to the data directory *on your computer*, probably:
data_dir= '/Users/Net_ID/data'

On Line #38
Set the variable prod_to_analyze to sst, using the line: **prod_to_analyze= 'sst'**

Save the text file…

2. Run the eof analysis program by typing:

   python class_eof_svdc.py

3. The program does not display the results directly, but rather writes the results out to the following directory:

   ~/data/sst_eof_output

4. Open the image **spatial_engenfunction_mode_1.png** and note the phase difference between north and south hemispheres in the first mode

5. Open the image **spatial_engenfunction_mode_2.png** and note the clear El Nino pattern in the second mode

6. Do you see other regions in the second mode that are *in phase* with the El Nino pattern in the Pacific (i.e., share a similar color to the El Nino signal)?

7. Do you see other regions in the second mode that are *out of phase* with the El Nino pattern in the Pacific (i.e., share a opposite colors to the El Nino signal in the spatial eigenfunction map)

8. Open the **class_eof_svdc.py** with a Sublime and in **Line 38**, change the product to analyze to chlorophyll with the following line of code:

    prod_to_analyze= 'chl'

9. The result from the chlorophyll analysis are output to:

    ~/data/chl_eof_output

10. Open the files in the output directory and make a special note of both the spatial eigenfunction maps AND the principle component time series.

11. Finally, open the **class_eof_svdc.py** with a text editor and have a look at the code and ask me any questions about what the code is doing…

NOTE: Mac Users --- You need to Install Apple's **XCode Developer's Application** and launch it manually at least once to register the software *before* installing SeaDAS.

Environmental Variables that need to be set in the *~/.bashrc* Unix startup file…

```
alias swget='wget load-cookies=~/.urs_cookies save-cookies=~/.urs_cookies auth-no-challenge=on --keep-session-cookies'
alias scurl='curl  -b ~/.urs_cookies -c ~/.urs_cookies -L -n -O'

export EDITOR=nano

# NOTE: ->  The two lines of code just below should be written as one single line of text in your .bashrc file.
alias swget='wget --load-cookies ~/.urs_cookies --save-cookies ~/.urs_cookies  --auth-no-challenge=on --keep-session-cookies'
alias scurl='curl -O -b ~/.urs_cookies -c ~/.urs_cookies -L -n'

export EDITOR=nano

export
PYTHONPATH=$PYTHONPATH:~/python_programs:~/python_programs/utilities:~/python_programs/tutorial_programs:~/python_programs/python_problem_set

export LOCAL_RESOURCES=~/python_programs/local_processing_resources

export MYPYTHONPROGRAMS=~/python_programs

# assumes you installed anaconda python in your home directory
export PATH="~/anaconda3/bin:$PATH"
```

NOTE: Mac Computers run the *~/.bash_profile* file instead of the *~/.bashrc* file. at Unix Terminal startup. Do the following to ensure the *~/.bashrc* file is run at Unix Terminal startup,

1. Open the ~/.bash_shell text file and add the following lines: source ~/.bashrc

2. You also need to change the default shell from the zsh to the bash shell. Go the System Preferences > Users and then unlock the padlock with your password. Right click on your username and click on *Advanced Options*. For Login Shell, choose /bin/bash

Wget Set Up

1. **Install via Homebrew Manager**: https://brew.sh
    a. Then install wget using homebrew by opening a Unix Terminal Window and typing the following:

    brew install wget
    brew upgrade wget

Create and/or Edit The ~/.netrc file:

1. Open (*or cerate*) the file: ***~/.netrc*** and then type the following lines:

   machine urs.earthdata.nasa.gov login *your_urs_username* password *your_urs_password*
   machine oceandata.sci.gsfc.nasa.gov login *your_urs_username* password *your_urs_password*
   machine ftp.ssmi.com login *your_remss_username* password *your_remss_password*
   machine ftp.remss.com login *your_remss_username* password *your_remss_password*

2. Be sure you are specifically registered for **NASA GESDISC DATA ARCHIVE**
   Become a registered user if you are not already a user
   https://urs.earthdata.nasa.gov

   Go to your user profile
   https://urs.earthdata.nasa.gov/profile
   Use the pulldown menu "My Applications"
   Select: "Approve More Applications"
   Choose and Approve **NASA GESDISC DATA ARCHIVE**

3. Add your **URS Earth Data username and password** to your ~/.netrc file for:

   machine urs.earthdata.nasa.gov login *your_urs_username* password *your_urs_password*
   machine oceandata.sci.gsfc.nasa.gov login *your_urs_username* password *your_urs_password*

4. **Create a blank cookie file**: At the terminal prompt, type**: touch ~/.urs_cookies**

5. To test that things are setup properly, trying getting a file using the following steps:
   a. Open a new Terminal Window
   b. Type: swget https://oceandata.sci.gsfc.nasa.gov/ob/getfile/A2021001000000.L1A_LAC.bz2

   If this starts the download of the satellite data file, you should be all set to go!

6. **NOTE:** If you run into problems retrieving data, try to take the following steps.

   If running on an Apple Mac OS…

   - Install MacPorts (https://www.macports.org/install.php) specific for your current OSX.
   - For MacPort installed openssl type the following to install the Certificate Authority:
   - **$ sudo port install curl-ca-bundle**
   - Then push its reference to the wget settings profile:
   - **$ echo CA_CERTIFICATE=/opt/local/share/curl/curl-ca-bundle.crt >> ~/.wgetrc**

   See: http://superuser.com/questions/262809/where-do-i-install-certificates-so-that-wget-and-other-macports-programs-will-fi
   See Also: http://andatche.com/blog/2012/02/fixing-ssl-ca-certificates-with-openssl-from-macports/

Python Environment/Modules with Anaconda

1. Go to the **Anaconda Website** and download that latest version of python.

2. Install and/or Update the following python packages by typing the following lines one-by-one in a Terminal Wondow:

   conda update conda
   conda config --add channels conda-forge
   conda install -c anaconda requests
   conda install matplotlib
   conda install -c anaconda netcdf4
   conda install -c cistools pyhdf
   conda install hdf5
   conda install pyproj
   conda install -c conda-forge pyresample
   conda install -c conda-forge cartopy
   conda install -c anaconda pillow
   conda update --all

   NOTE: In some computer labs, the following additional packages are needed
   conda install -c anaconda certifi
   conda install -c anaconda ncurses

   certifi allows some anaconda packages (e.g. cartopy) to retrieve additional components on the fly (e.g., high resolution coastlines)
   ncurses allows arrow backspace to work in when using python from a terminal.

3. Open your **~/.bashrc file** and add the following line near the bottom of the text file:

   export PATH="/Users/bmonger/anaconda3/bin:$PATH"

   # **where** you should relace the Ancacond path above with the path on your own computer.
   # To find the Anaconda path: type the following in a new terminal window:

   which python

Installing SeaDAS

1. **FIRST**, be sure that wget is properly configured to access https (*see above*).

2. GoTo: https://seadas.gsfc.nasa.gov
   a. Click On: Download
   b. Download SeaDAS Visualization Installer Script for your machine.
   c. Move the script to your Home Directory
   d. Open a Terminal Window
      i. Make the Installer Script "executable" by typing:
      sudo chmod -x *the-name-of-your-installer-script*
      ii. Follow the GUI Instructions.
      I like placing seadas in my Applications directory I use the destination directory: **/Applications/seadas-8.3**

   NOTE: that this step only installs the data visualization capabilities. Additional steps are needed to download satellite data processing capabilities.

3. Scroll Down the Download the SeaDAS Download Web Page until you get to the *Manual Installation Instructions*.

   a) Ensure that machine requirements are met
   b) Download the Installer Script (named: install_ocssw)
   c) Download the Manifest Phyton Program and place in the ~/python_programs/utilities folder.
   d) Make sure it is executable by typing: sudo chmod +x install_ocssw.py
   e) Check for the latest version of processing modules by typing:

      ./install_ocssw --list_tags

   f) NOTE: If the command above give a tag **V**####.# greater than **V**2023.0, then use the larger tag version in place of V2023.2 below. Or try **T2023.9 (beta version that seems best for the Mac OS right now).**
   g) Run the installer script (***as a single line in the Terminal window***)…

```
./install_ocssw --install_dir /Applications/seadas-8.3/ocssw
--seadas --tag T2023.9 --seawifs --modisa --modist --viirsn
--hico  --meris --olcis3a --olcis3b  --msis2a --msis2b
--viirsj1 --viirsdem --oci --ocis --hawkeye
```

   a. Close and reopen the Terminal Window

4. Check that your **~/.bashrc file** has the following lines:

   export OCSSWROOT=/Applications/seadas-8.1/ocssw
   source $OCSSWROOT/OCSSW_bash.env

5. Open up the write permissions for automatic ancillary data downloads that occur during ocean color processing.

   sudo chmod -R 777 /Applications/seadas-8.1/ocssw/

6. Update the lookup tables for each satellite sensor using the script *update_luts*. Get help on syntax by simply typing the following at the terminal prompt: update_luts *<return>*

7. There is an executable file called ***seadas*** located in: /Applications/seadas-8.1/bin/ that should be dragged to the Desktop for easy access.

Set up a *Cron Job* to update SeaDAS Lookup tables daily

A cron job is a script that your computer will automatically run at specific time intervals. We will need to set up daily cron jobs to regularly update SeaDAS lookup table. To set up these cron jobs, do the following:

1. Launch Terminal
2. Open a cron job script in nano, which is the default text editor we set up when we altered the bash profile. Do this by typing: **nano crontab -e <RETURN>**
3. Type the following into the text editor, again be careful about white space:

   0 0 * * * /bin/csh -c "$OCSSWROOT/bin/update_luts all"

4. Exit the text editor by typing **CNTRL X**
5. Save to buffer with the default filename by typing **Y <RETURN>**