Your First Python Tutorial for Scientists

/ to
non
will
/sis

and

visualizations. Python can be compared to other high-level, interpreted, object-oriented languages, but is especially great because it is free and open source!

High level languages:

Other high level languages include MatLab, IDL, and NCL. The advantage of high level languages is that they provide functions, data structures, and other utilities that are commonly used, which means it takes less code to get real work done. The disadvantage of high level languages is that they tend to obscure the low level aspects of the machine such as: memory use, how many floating point operations are happening, and other information related to performance. C and C++ are all examples of lower level languages. The "higher" the level of language, the more computing fundamentals are abstracted.

Interpreted languages:

Most of your work is probably already in interpreted languages if you've ever used IDL, NCL, or MatLab (interpreted languages are typically also high level). So you are already familiar with the advantages of this: you don't have to worry about compiling or machine compatability (it is portable). And you are probably familiar with their deficiencies: sometimes they can be slower than compiled languages and potentially more memory intensive.

Object Oriented languages:

Objects are custom datatypes. For every custom datatype, you usually have a set of operations you might want to conduct. For example, if you have an object that is a list of numbers you might want to apply a mathematical operation, such as sum, onto this list object in bulk. Not every function can be applied to every datatype; it wouldn't make sense to apply a logarithm to a string of letters or to capitalize a list of numbers. Data and the operations applied to them are grouped together into one object.

Open source:

Python as a language is open source which means that there is a community of developers behind its codebase. Anyone can join the developer community and contribute to deciding the future of the language. When someone identifies gaps to Python's abilities, they can write up the code to fill these gaps. The open source nature of Python means that Python as a language is very adaptable to shifting needs of the user community.

Python is a language designed for rapid prototyping and efficient programming. It is easy to write new code quickly with less typing.

Why another Python tutorial?

What makes this Python tutorial unique is that it has been designed specifically to meet the needs of, and feedback from, atmospheric and oceanic scientists making the transition with the NCAR-wide pivot-to-Python. In particular, this tutorial should be useful to any scientist who already knows how to program in some other language but is taking up Python for the first time. By spending the first course on pure Python without importing any additional packages, our "Your First" tutorial addresses the concerns that most tutorials either pick up speed too quickly by going into the intricacies of third-party packages before explaining how Python is different from other languages, or get too bogged down in basic programming concepts that anyone with programming experience already knows. This tutorial attempts to hit the sweet spot between too high-level and too low-level. By using coding examples with real atmospheric datasets and questions, the skills and techniques taught are easily applied to actual atmospheric or oceanic workflows. We hope that this tailored approach to teaching and sharing computational tools effectively addresses the concerns and needs of the geoscience community.

See also

- Official Python 3 Documentation
- Official GitHub Documentation
- Official Git Documentation

Requirements & Installation

We will be using the Conda package manager in this tutorial. If you don't have Conda installed at all, please install it. Conda is an excellent package manager for Python development, but it is capable of full end ion ent e a will

1. Check that you have conda or miniconda installed on your OS by checking your conda version:

```
$ conda --version
```

   At the time of writing this, the latest version of conda is 4.8. If you have an old version of conda installed, update it.

2. If necessary, update:

```
$ conda update -n base conda
```

   Updating your Conda package manager should not have any effect on your existing Conda environments.

   Note

   If you have a *really* old version of conda it might be easier to delete it and then reinstall it. But before doing this you have to check your env-list with `conda env list` to see if there are any environments you created and want to save.

3. Check your conda version again.

```
$ conda --version
```

4. Initialize Conda to work with your shell (.e.g., `bash`):

```
$ conda init
```

   This step may modify your shell configuration script (e.g., `.bash_profile`) to make the `conda` command available in your shell, and it will make the `conda activate` command work.

5. Install and Configure Git

   Git is a program that tracks changes made to files. This makes it easy to maintain access to multiple versions of your code as you improve it, and revert your code back to a previous version if you've made any mistakes.

First Python Script

This section of the tutorial will focus on teaching you Python through the creation of your first script. You will learn about syntax and the reasoning behind why things are done the way they are along the way. We will also incorporate lessons on the use of Git because we highly recommend you version controling your work.

We are assuming you are familiar with bash and terminal commands. If not here is a cheat sheet.

Reading a .txt File

In building your first Python script we will set up our workspace, read a `.txt` file, and learn Git fundamentals.

Open a terminal to begin.

Note

l it

2. Go into the directory:

```
$ cd python_tutorial
```

3. Create a virtual environment for this project:

```
$ conda create --name python_tutorial python
```

A conda environment is a directory that contains a collection of packages or libraries that you would like installed and accessible for this workflow. Type `conda create --name` and the name of your project, here that is `python_tutorial`, and then specify that you would like to install Python in the virtual environment for this project.

It is a good idea to create new environments for different projects because since Python is open source, new versions of the tools you use may become available. This is a way of guaranteeing that your script will use the same versions of packages and libraries and should run the same as you expect it to.

See also

More information on Conda environments

4. And activate your Conda environment:

```
$ conda activate python_tutorial
```

5. Make the directory a Git repository:

```
$ git init .
```

A Git repository tracks changes made to files within your project. It looks like a `.git/` folder inside that project.

This command adds version control to this new python_tutorial directory and all of its contents.

See also

More information on Git repositories

6. Create a data directory:

```
$ mkdir data
```

And we'll make a directory for our data.

7. Go into the data directory:

```
$ cd data
```

8. Download sample data from the CU Boulder weather station:

```
$ curl -kO https://sundowner.colorado.edu/weather/atoc8/wxobs20170821.txt
```

This weather station is a Davis Instruments wireless Vantage Pro2 located on the CU-Boulder
is
the

in

that directory) is listed as "untracked," which means all of the files you added to that directory are *also* untracked by Git. The `git status` command will tell you what to do with untracked files. Those instructions mirror the next 2 steps:

10. Add the file to the Git staging area:

```
$ git add wxobs20170821.txt
```

By adding this datafile to your directory, you have made a change that is not yet reflected in our Git repository. Every file in your working directory is classified by git as "untracked", "unmodified", "modified", or "staged." Type `git add` and then the name of the altered file to stage your change, i.e. moving a file that is either untracked or modified to the staged category so they can be committed.

See also

More information on git add

11. Check your git status once again:

```
$ git status
```

Now this file is listed as a "change to be commited," i.e. staged. Staged changes can now be commited to your repository history.

12. Commit the file to the Git repository:

```
$ git commit -m "Adding sample data file"
```

With `git commit`, you've updated your repository with all the changes you staged, in this case just one file.

Note

On a Windows machine you may see the following: `warning: LF will be replaced by CRLF.` The file will have its original line endings in your working directory`. Do not worry too much about this warning. CR refers to "Carriage Return Line Feed" and LF refers to "Line Feed." Both are used to indicate line termination. In Windows both a Carriage Return and Line Feed are required to note the end of a line, but in Linux/UNIX only a Line Feed is required. Most text editors can account for line ending differences between opperating systems, but sometimes a conversion is necessary. To silence this warning you can type `git config --global core.autocrlf false` in the terminal.

13. Look at the Git logs:

```
$ git log
```

If you type `git log` you will show a log of all the commits, or changes made to your repository.

14. Go back to the top-level directory:

```
$ cd ..
```

15. And now that you've set up our workspace, create a blank Python script, called `mysci.py`:

16. Edit the `mysci.py` file using nano, vim, or your favorite text editor:

```
1 print("Hello, world!")
```

Your classic first command will be to print `Hello, world!`.

> **Note**
>
> On a Windows machine, it is possible *nano* or *vim* are not recognized as text editors within your terminal. In this case simply try to run *mysci.py* to open a notepad editor.

17. Try testing the script by typing `python` and then the name of your script:

```
$ python mysci.py
```

Yay! You've just created your first Python script.

18. You probably won't need to run your Hello World script again, so delete the `print("Hello, world!")` line and start over with something more useful - we'll read the first 4 lines from our datafile.

    Change the `mysci.py` script to read:

```
1
2
3  # Read the data file
4  filename = "data/wxobs20170821.txt"
   datafile = open(filename, 'r')
5
   print(datafile.readline())
   print(datafile.readline())
6  print(datafile.readline())
   print(datafile.readline())
7
   datafile.close()
8
9
10
```

First create a variable for your datafile name, which is a string - this can be in single or double quotes.

Then create a variable associated with the opened file, here it is called `datafile`.

The `'r'` argument in the open command indicates that we are opening the file for reading capabilities. Other input arguments for open include `'w'`, for example, if you wanted to write to the file.

The readline command moves through the open file, always reading the next line.

And remember to close your datafile.

Comments in Python are indicated with a hash, as you can see in the first line `# Read the data file`. Comments are ignored by the interpreter.

ute
our

script.

20. Change the `mysci.py` script to read your whole data file:

```
# Read the data file
filename = "data/wxobs20170821.txt"
datafile = open(filename, 'r')

data = datafile.read()

datafile.close()

# DEBUG
print(data)
print('data')
```

Our code is similar as before, but now we've read the entire file. To test that this worked. We'll `print(data)`. Print statements in python require parenthesis around the object you wish to print, in this scenario the data object.

Try `print('data')` as well. Now Python will print the string `data`, as it did for the hello world function, instead of the information stored in the variable data.

Don't forget to execute with `python mysci.py`.

21. Change the `mysci.py` script to read your whole data file using a context manager with:

```
# Read the data file
filename = "data/wxobs20170821.txt"
with open(filename, 'r') as datafile:
   data = datafile.read()

# DEBUG
print(data)
```

Again this is a similar method of opening the datafile, but we now use `with open`. The `with` statement is a context manager that provides clean-up and assures that the file is automatically closed after you've read it.

The indendation of the line `data = datafile.read()` is very important. Python is sensitive to white space and will not work if you mix spaces and tabs (Python does not know your tab width). It is best practice to use spaces as opposed to tabs (tab width is not consistent between editors).

Combined these two lines mean: with the datafile opened, I'd like to read it.

And execute with `python mysci.py`.

See also

More information on context managers

22. What did we just see? What is the data object? What type is data? How do we find out?

    the datatype when you name a variable, Python will automatically figure it out by the nature of the data.

23. Now, clean up the script by removing the DEBUG section, before we commit this to Git.

24. Let's check the status of our Git repository

```
$ git status
```

Note

Take a look at which files have been changed in the repository!

25. Stage these changes:

```
$ git add mysci.py
```

26. Let's check the status of our Git repository,again. What's different from the last time we checked the status?

```
$ git status
```

27. Commit these changes:

```
$ git commit -m "Adding script file"
```

    Here a good commit message `-m` for our changes would be `"Adding script file"`

28. Let's check the status of our Git repository, now. It should tell you that there are no changes made to your repository (i.e., your repository is up-to-date with the state of the code in your directory).

```
$ git status
```

29. Look at the Git logs, again:

```
$ git log
```

    You can also print simplified logs with the `--oneline` option.

That concludes the first lesson of this virtual tutorial.

In this section you set up a workspace by creating your directory, conda environment, and git repository. You downloaded a .txt file and read it using the Python commands of `open()`, `readline()`, `read()`, `close()`, and `print()`, as well as the context manager `with`. You should be familiar with the `str` datatype. You also used fundamental git commands such as `git init`, `git status`, `git add`, `git commit`, and `git log`.

See also

our
o a
out

1. One big string isn't very useful, so use `str.split()` to parse the data file into a data structure you can use.

   Change the `mysci.py` script to read:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
# Initialize my data variable
data = []

# Read and parse the data file
filename = "data/wxobs20170821.txt"
with open(filename, 'r') as datafile:

 # Read the first three lines (header)
 for _ in range(3):
    datafile.readline()

 # Read and parse the rest of the file
 for line in datafile:
    datum = line.split()
    data.append(datum)

# DEBUG
for datum in data:
   print(datum)
```

The first thing that is different in this script is an initialized data variable; `data = []` creates the variable data as an empty `list` which we will populate as we read the file. Python `list` objects are a collection data type that contain ordered and changeable - meaning you can call information out of the `list` by its index and you can add or delete elements to your `list`. Lists are denoted by square brackets, `[]`.

Then with the datafile open for reading capabilities, we are going to write two separate `for` loops. A `for` loop is used for iterating over a sequence (such as a list). It is important to note the syntax of Python `for` loops: the `:` at the end of the `for` line, the tab-indentation of all lines within the `for` loop, and perhaps the absence of an `end for` that is found in languages such as Matlab.

In your first `for` loop, loop through the dummy variable `_` in `range(3)`. The `range` function returns a sequence of numbers, starting at 0 and incrementing by 1 (by default), ending at the specified length. Here if you were to `print(_)` on each line of the for loop you would see:

```
0
1
2
```

the

ber

the

ach

n a

but

other useful options are `/t` for splitting along tabs or `,` along commas.

Then you `append` this split line list to the end of your data `list`. The `list.append()` method adds a single item to the end of your `list`. After every line in your `for` loop iteration, the data `list` that was empty is one element longer. Now we have a `list` of `list`s for our data variable - a `list` of the data in each line for multiple lines.

When you print each datum in data, you'll see that each datum is a `list` of `string` values.

We just covered a lot of Python nuances in a very little bit a code!

See also

More information on for-loops More information on Python lists

2. Now, to practice list indexing, get the first, 10th, and last row in data.

   Change the DEBUG section of our `mysci.py` script to:

```
17
18  # DEBUG
19  print(data[0])
    print(data[9])
    print(data[-1])
20
```

   Index your list by adding the number of your index in square brackets, `[]`, after the name of the `list`. Python is 0-indexed so `data[0]` refers to the first index and `[-1]` refers to the last index.

3. Now, to practice slice indexing, get the first 10 rows in data.

   Change the DEBUG section of our `mysci.py` script to:

```
17
18  # DEBUG
    for datum in data[0:10]:
19      print(datum)
```

   Using a colon, `:`, between two index integers `a` and `b`, you get all indexes between `a` and `b`. See what happens when you print `data[:10]`, `data[0:10:2]`, and `data[slice(0,10,2)]`. What's the difference?

4. Now, to practice nested indexing, get the 5th, the first 5, and every other column of row 9 in the data object.

   Change the DEBUG section of the `mysci.py` script to:

```
17
18  # DEBUG
    print(data[8][4])
19  print(data[8][:5])
    print(data[8][::2])
20
```

In nested `list` indexing, the first index determines the row, and the second determines the element from that row. Also try printing `data[5:8][4]`, why doesn't this work?

ıich

ole,

hat happens when you try to do something like `data[:][4]`!)

Let's get our data into a more convenient named-column format.

Change `mysci.py` to the following:

```
1
2
3
4
5
6   # Initialize my data variable
    data = {'date': [],
7     'time': [],
      'tempout': []}
8
9   # Read and parse the data file
    filename = "data/wxobs20170821.txt"
10  with open(filename, 'r') as datafile:

11     # Read the first three lines (header)
       for _ in range(3):
12        datafile.readline()

13     # Read and parse the rest of the file
       for line in datafile:
14        split_line = line.split()
          data['date'].append(split_line[0])
15        data['time'].append(split_line[1])
          data['tempout'].append(split_line[2])
16
17  # DEBUG
    print(data['time'])
18
19
20
21
22
```

First we'll initialize a dictionary, `dict`, indicated by the curly brackets, `{}`. Dictionaries, like `list`s, are changeable, but they are unordered. They have keys, rather than positions, to point to their elements. Here you have created 3 elements of your dictionary, all currently empty `list`s, and specified by the keys `date`, `time`, and `tempout`. Keys act similarly to indexes: to pull out the `tempout` element from data you would type `data['tempout']`.

Grab date (the first column of each line), time (the second column of each line), and temperature data (the third column), from each line and `append` it to the `list` associated with each of these data variables.

See also

More on Python dictionaries

7. Clean up (remove DEBUG section), stage, and commit

I in
e is

```
data['tempout'].append(split_line[2])
```

to:

```
19  data['tempout'].append(float(split_line[2]))
```

The `float` datatype refers to floating point real values - the datatype of any numbers with values after a decimal point. You could also change the datatype to `int`, which will round the values down to the closest full integer.

See also

More on Python numeric types (int, float, complex)

9. Add a DEBUG section at the end and see what `data['tempout']` now looks like.

   Do you see a difference? It should now be a list of floats.

10. Clean up (remove DEBUG section), stage, and commit

```
$ git add mysci.py
$ git commit -m "Converting tempout to floats"
```

11. This seems great, so far! But what if you want to read more columns to our data later? You would have to change the initialization of the data variable (at the top of `mysci.py`) and have to add the appropriate line in the "read and parse" section. Essentially, that means you need to maintain 2 parts of the code and make sure that both remain consistent with each other.

    This is generally not good practice. Ideally, you want to be able to change only one part of the code and know that the rest of the code will remain consistent. So, let's fix this.

Change `mysci.py` to:

```
1
2
```

```
10 # Data types for each column (only if non-string)
   types = {'tempout': float}
11
   # Initialize my data variable
12 data = {}
   for column in columns:
13    data[column] = []

14 # Read and parse the data file
   filename = "data/wxobs20170821.txt"
15 with open(filename, 'r') as datafile:

16    # Read the first three lines (header)
      for _ in range(3):
17       datafile.readline()

18    # Read and parse the rest of the file
      for line in datafile:
19       split_line = line.split()
         for column in columns:
20          i = columns[column]
            t = types.get(column, str)
21          value = t(split_line[i])
            data[column].append(value)
22
   # DEBUG
23 print(data['tempout'])
24
25
26
27
28
29
30
```

You have now created a columns `dict` that points each data variable to its column-index. And a types `dict`, that indicates what type to convert the data when necessary. When you want new variables pulled out of the datafile, change these two variables.

Initializing the data `dict` now includes a `for` loop, where for each variable specified in columns, that key is initialized pointing to an empty `list`. This is the first time you have looped over a `dict` and added key-value pairs to a `dict` via assignment.

When reading and parsing the file, you created your first nested `for` loop. For every line of the datafile, split that line - and then for every desired variable in the columns `dict` (date, time, tempout): grab the datum from the current split line with the specified index (0, 1, 2), use the `dict.get()` method to find the desired datatype if specified (avoiding `key-not-found` errors and defaulting to `str` if unspecified), convert the datum to the desired datatype, and `append` the datum to the `list` associated with each column key within the data `dict`.

12. Clean up (remove DEBUG section), stage, and commit

```
$ git add mysci.py
$ git commit -m "Refactoring data parsing code"
```

That concludes the second lesson of this virtual tutorial.

In this section you saved the variables of date, time, and tempout in a data dictionary.

and
You
nds

Writing Functions

This is intended to pick off right where "Creating a Data Dictionary" left off - you had just commited your new script that reads the file, saving the variables of date, time, and tempout in a data dictionary. In this section you will compute wind chill index by writing your first function and learning about basic math operators.

Let's begin.

1. Okay, now that you've read the data in a way that is easy to modify later, it is time to actually do something with the data.

   Compute the wind chill factor, which is the cooling effect of the wind. As wind speed increases the rate at which a body loses heat increases. The formula for this is:

   $$WCI = a + (b * t) - (c * v^2) + (d * t * v^2)$$

   Where *WCI* refers to the Wind Chill in degrees F, *t* is temperature in degrees F, *v* is wind speed in mph, and the other variables are as follows: *a* = 35.74, *b* = 0.6215, *c* = 35.75, and *d* = 0.4275. Wind Chill Index is only defined for temperatures within the range -45 to +45 degrees F.

   You've read the temperature data into the tempout variable, but to do this calculation, you also need to read the windspeed variable from column 7.

   Modify the columns variable to read:

   ```
   # Column names and column indices to read
   columns = {'date': 0, 'time': 1, 'tempout': 2, 'windspeed': 7}
   ```

   and modify the types variable to be:

   ```
   # Data types for each column (only if non-string)
   types = {'tempout': float, 'windspeed': float}
   ```

2. Great! Save this in your Git repo. Stage and commit

   ```
   $ git add mysci.py
   $ git commit -m "Reading windspeed as well"
   ```

3. Now, let's write our first function to computethe wind chill factor. We'll add this function to the bottom of the file.

```
29
30
31
38
```

To indicate a function in python you type `def` for define, the name of your function, and then in parenthesis the input arguments of that function, followed by a colon. The preceding lines,the code of your function, are all tab-indented. If necessary specify your return value.

See also

More on user defined functions

Here is your first introduction to math operators in Python. Addition, subtraction, and multiplication look much like you'd expect. A double astericks, `**`, indicates an exponential. A backslash, `/`, is for division, and a double backslash, `//`, is for integer division.

And then let's compute a new list with windchill data at the bottom of `mysci.py`:

```
40
41  # Let's actually compute the wind chill factor
42  windchill = []
    for temp, windspeed in zip(data['tempout'], data['windspeed']):
43     windchill.append(compute_windchill(temp, windspeed))
```

Now we'll call our function. Initialize a `list` for wind chill with empty square brackets, `[]`. And in a `for` loop, loop through our temperature and wind speed data, applying the function to each `tuple` data pair. `tuple`s are ordered like `list`s, but they are indicated by parenthesis, `()`, instead of square brackets and cannot be changed or appended. `tuple`s are generally faster than `list`s.

We use the `zip` function in Python to automatically unravel the `tuple`s. Take a look at `zip([1,2], [3,4,5])`. What is the result?

And finally, add a DEBUG section to see the results:

```
45
46  # DEBUG
    print(windchill)
```

4. Clean up, stage, and commit

```
$ git add mysci.py
$ git commit -m "Compute wind chill factor"
```

5. Now, the wind chill factor is actually in the datafile, so we can read it from the file and compare that value to our computed values. To do this, we need to read the windchill from column 12 as a `float`:

   Edit the columns and types `dict`:

```
1  # Column names and column indices to read
2  columns = {'date': 0, 'time': 1, 'tempout': 2, 'windspeed': 7,
3             'windchill': 12}
```

Note

ne

om

```
46
47
48
```

```
# DEBUG
for wc_data, wc_comp in zip(data['windchill'], windchill):
    print(f'{wc_data:.5f}    {wc_comp:.5f}    {wc_data - wc_comp:.5f}')
```

Using `f-string`s with float formatting you can determine the precision with which to print the values to. The `:python:.5f` means you want 5 places after the decimal point.

See also

More on string formatting

Test the results. What do you see? Our computation isn't very good is it?

6. Clean up, stage, and commit

```
$ git add mysci.py
$ git commit -m "Compare wind chill factors"
```

7. Now, format the output so that it's easy to understand and rename this script to something indicative of what it actually does.

   To the end of the file, add:

```
46
47
48
49
50
51
52
53
```

```
# Output comparison of data
print('                    ORIGINAL  COMPUTED')
print(' DATE    TIME  WINDCHILL WINDCHILL DIFFERENCE')
print('------- ------ --------- --------- ----------')
zip_data = zip(data['date'], data['time'], data['windchill'], windchill)
for date, time, wc_orig, wc_comp in zip_data:
    wc_diff = wc_orig - wc_comp
    print(f'{date} {time:>6} {wc_orig:9.6f} {wc_comp:9.6f} {wc_diff:10.6f}')
```

   Here you used *f-string* formatting with more *f-string* formatting options. The `>6` indicates that you'd like the characters of the string to be right-justified and to take up 6 spaces.

   The `9f` specifies that you want the value to fill 9 spaces, so `9.6f` indicates you'd like the value to fill 9 spaces with 6 of them being after the decimal point. Same concept for `10.6f`.

   You now have your first complete Python script!

8. DON'T CLEAN UP! Just stage and commit

```
$ git add mysci.py
$ git commit -m "Output formatting comparison data"
```

9. Let's rename this script to something meaningful and indicative of the computation inside.

```
$ git mv mysci.py windchillcomp.py
$ git commit -m "Renaming first script"
```

our

lew

and

GitHub repository to share a name.

4. And click "Create Repository"

5. Copy the link to your GitHub repository.

   Copy the link in the input right beneath the title, it should look something like this:

   `https://github.com/<user_name>/<repo>.git`

6. Then to set your remote repository, in your project terminal type:

```
$ git remote add origin <remote repository URL>
```

> **Note**
>
> Your remote repository URL is the link you copied in step 5!

7. And verify your remote repository:

```
$ git remote -v
```

8. And finally push your project to GitHub:

```
$ git push origin master
```

Think of GitHub as online storage for versions of your project, much like hosting your code in a Google Drive, but with better features specific to coding. A lot of GitHub's features show their usefulness when you are working collaboratively, sharing your code with other scientists, or if you wanted to display and easily visualize changes in your code between commits.

That concludes the "First Python Script" virtual tutorial where you learned to write your first Python script.

In this section you calculated wind chill index by writing and calling your first function. You also learned about Python math operators, the `zip()` command, `tuple` datastructure, *f*-`string` formatting, and how to push your repository to GitHub.

> **See also**
>
> - User defined functions
> - String formatting

This material is based upon work supported by the National Center for Atmospheric Research, a major facility sponsored by the National Science Foundation and managed by the University Corporation for Atmospheric Research. Any opinions, findings and

conclusions or recommendations expressed in this material do not necessarily reflect the views of the National Science Foundation.