Climate Data Analysis Tools (CDAT): A Beginner's Guide

Version 3.2

PCMDI Computational Support

Program for Climate Model Diagnosis and Intercomparison (PCMDI)
Lawrence Livermore National Laboratory
Livermore, CA 94550
United States of America

http://cdat.sf.net

6/3/02

Legal Notice

This work was produced at the University of California, Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48 between the U.S. Department of Energy and The Regents of the University of California for the operation of UC LLNL.

Table of Contents

CHAPTER 1 *CDAT* ***1***

Introduction **1**
Downloading CDAT. **2**
Installing CDAT. **3**
System Requirements **3**
How to use this guide. **3**

CHAPTER 2 *Getting Started* ***5***

Try the Visual CDAT browser first **5**
Description of the VCDAT Interface ***7***
A brief introduction to Python scripting. **16**
Variables and arithmetic expressions: ***17***
Conditional statements: ***19***
File Input and Output: ***20***
Lists and Tuples: ***20***
Loops: ***22***
Dictionaries: ***23***
Functions: ***23***
Modules: ***24***
How to get the tutorials and data **25**
What do the tutorials cover? **25**
getting_started_tutorial.py ***26***
times_tutorial.py ***26***
statistics_tutorial.py ***27***
vcs_tutorial.py ***27***
xmgrace_tutorial.py ***27***
Documentation and Help **27**
Online Documentation ***28***
Printable documentation: ***29***
Getting Support ***30***

CHAPTER 3 *What's in CDAT* ***31***

File/Data Handling **31**

File I/O ***31***

Reading, creating, and altering variables ***33***

Handling Time ***41***

Axes and Domains ***43***

Data Selection. ***44***

Regridding Data ***45***

Working with masks ***46***

Databases ***47***

Averaging and Statistics **48**

Area averaging ***48***

Generating weights ***50***

Time averaging ***51***

Useful statistical functions ***55***

Data Visualization **57**

Visualization and Control System (VCS) ***58***

Displaying data ***59***

Interface to Grace (genutil.xmgrace) ***67***

Other useful packages **68**

Interface to Spherepack ***68***

Interface to Regridpack ***68***

Empirical Orthogonal Functions ***68***

Interface to the L-moments library ***69***

Interface to the ngmath library ***69***

Using existing Fortran code ***69***

Migrating from GrADS (grads) ***70***

ort ***70***

trends ***71***

CHAPTER 4 *Contributions to CDAT* ***73***

How to add your packages **73**

Index 75

CHAPTER 1

CDAT

1.1 Introduction

Climate Data Analysis Tools (CDAT) is a software infrastructure that uses an object-oriented scripting language to link together separate software subsystems and packages, thus forming an integrated environment for solving model diagnosis problems. The power of the system comes from Python and its ability to seamlessly interconnect software. Python provides a general purpose and full-featured scripting language with a variety of user interfaces including command-line interaction, stand-alone scripts (applications) and graphical user interfaces (GUI). The CDAT subsystems, implemented as modules, provide access to and management of gridded data (Climate Data Management System or CDMS); large-array numerical operations (Numerical Python); and visualization (Visualization and Control System or VCS).

One of the most difficult challenges facing climate researchers today is the cataloging and analysis of massive amounts of multi-dimensional global atmospheric and oceanic model data. To reduce the labor intensive and time-consuming process of data management, retrieval, and analysis, PCMDI and other DOE sites have come together to develop intelligent filing system and data management software for the linking of storage devices located throughout the United States and the international climate research community. This effort, headed by PCMDI, NCAR, and ANL will allow users anywhere to remotely access this distributed multi-petabyte archive and perform analysis. PCMDI's CDAT is an innovative system that supports exploration and visualization of climate scientific datasets. As an "open system", the software sub-systems (i.e., modules) are independent and freely available to the global climate community. CDAT is easily extended to include new modules and as a result of its flexibility, PCMDI has integrated other popular software components, such as: the popular Live Access Server (LAS) and the Distributed Oceanographic Data System (DODS). Together with ANL's Globus middleware software, CDAT's focus is to allow climate researchers the ability to access and analyze multi-dimensional distributed climate datasets.

1.2 Downloading CDAT.

CDAT can be downloaded from the **CDAT Home Page** at **http://cdat.sf.net** as a gzipped tar file (cdat.tar.gz). Links to the latest source can be found there. If you are using CDAT for the first time, it is recommended that you also download the tutorials and tutorial data from the CDAT home page at http://cdat.sf.net. The tutorial files are in **cdat_tutorial.tar.gz** and the data files used in the tutorials are in **cdat_tutorial_data.tar.gz**.

1.3 Installing CDAT.

The downloadable tar file contains a README.TXT file which lists the requirements and instructions to install CDAT. The instructions are not listed here so that this document is not outdated when new versions of CDAT are released.

1.4 System Requirements

CDAT has been successfully installed on a variety of platforms. The specific platforms it has been tested on include Sun/Solaris (5.6), DEC OSF 1 (5.0), Irix (6.5), HP-UX 11.0 (We know 10 will not work), MacOSX "Darwin" and various versions of Linux (Kernel 2.2 or greater). The source distribution of CDAT includes packages required by CDAT and the installation script handles the task of installing them in the appropriate areas.

1.5 How to use this guide.

This guide is intended to provide users who are new to CDAT a single document where they can get an overall view of CDAT and to provide pointers to more detailed information. There are numerous packages that comprise CDAT. To document all their features here would make this document unreasonably large and daunting to the new user. Instead, the approach adopted here is to give a brief overview of how tasks are accomplished in CDAT and point the reader to other documents or methods of getting additional documentation should they require more details. In that sense this is also a “Documentation of Available Documentation”.

Chapter 2 contains instructions on getting started with using CDAT. The Graphical User Interface, VCDAT (which stands for Visual

Climate Data Analysis Tools), is introduced in this chapter. It is a powerful tool to browse through data, perform data analysis, find documentation, and to learn the scripting and other capabilities of CDAT. However, CDAT does not require that you use the graphical interface and it is easily executed from the Python command line. A brief introduction to Python is included in this section. The tutorials have example scripts and a set of example datasets. These tutorial scripts are easy to run and provide a range of real life applications that a climate scientist can see in action. The various methods of getting the necessary documentation and help in using CDAT, ranging from printable files to online documentation, FAQs, and discussion groups are listed in this chapter.

Having gone through the tutorial, the next step is to become aware of CDAT's features. Chapter 3 entitled "What's in CDAT?" addresses this. To illustrate the features of CDAT, we have tried to address some of the tasks that a climate scientist may wish to accomplish. Therefore, the organization of tasks is broken up along the lines of File I/O, creating databases and accessing data from them, data extraction, altering variables and metadata, regridding data, spatial and temporal averaging, statistics, and visualization. In addition to these, other sample scripts which describe such tasks as making use of exiting Fortran or C code, and interfacing to specialized packages such as spherepack and EOFs, provide the reader with a flavor for what is already possible to do with CDAT and also how easy it is to leverage off previous efforts. The final chapter in this guide explains how you can contribute to CDAT.

CHAPTER 2 Getting Started

CDAT uses the Python language as a glue to link together the various tools. Python has one of the fastest growing user bases among programming languages today and is extensively documented online and in publications. It must be stressed that it is not essential to have any knowledge of Python before being able to use CDAT tools. The Graphical User Interface (VCDAT) can be used with no knowledge of Python and is the first topic introduced in this chapter. This is highly recommended for beginners. For the more adventurous beginner who wishes to learn the use of scripting capabilities, a brief background of the Python language and syntax is provided in the next section.

CDAT comes with a suite of tutorials to help you learn how to use it. These tutorials are described in brief to enable the user to find the appropriate tutorials and to give a flavor of the power of CDAT. Finally, the various documentation and help avenues available to the user are described in this chapter.

2.1 Try the Visual CDAT browser first

Before you get into the tutorials, try the VCDAT browser first. This is accomplished by simply typing the command “vcdat” at your shell prompt, i.e:

```
% vcdat
```

VCDAT was designed to be used from left-to-right and top-to-bottom and has on-line help balloons to assist the user in navigating through the interface. That is, if the user becomes unsure of what to do at any give time, then by moving the mouse over the region of question and letting it rest will result in a help balloon popup with information to assist the user. VCDAT allows the user to enter command line instructions and logs most button click instructions in a script file for later reference. Although it is not required to learn CDAT scripting in order to use VCDAT, the interface can be used as a CDAT script-learning tool by translating every button press and keystroke into a script file. All scripts include comment lines to assist the user. The script file can be modified, saved, and executed by the user. Thus, helping the user learn how to read and write CDAT scripts. This facility also allows the non-interactive repetition of common tasks.

To demonstrate quickly, the use of VCDAT try the following example sequence of button clicks:

Example 1: To quickly browse and plot data:

1. Select the directory where the data is stored,
2. Select the desired file that is located in the directory,
3. Select a variable from the file,
4. Click the "Plot" button.

Example 2: To select data ranges and plot type:

1. Select the directory where the data is stored,
2. Select the desired file that is located in the directory,
3. Select a variable from the file,
4. Select variable ranges (i.e., experiment by adjusting the horizontal slider bars located just below the "Plot" button,
5. Select graphics method (e.g., select "Isofill" in the choice button located to the right of the "Plot" button,
6. Click the "Plot" button.

Example 3: To average over a dimension:

1. Select the directory where the data is stored,
2. Select the desired file that is located in the directory,

3. Select a variable from the file,
4. Select "avg" if applicable (e.g., to the right of the horizontal slider bars, which are located below the "Plot" button, are choice buttons displaying "def". Select "avg" in place of "def".),
5. Click the "Plot" button.

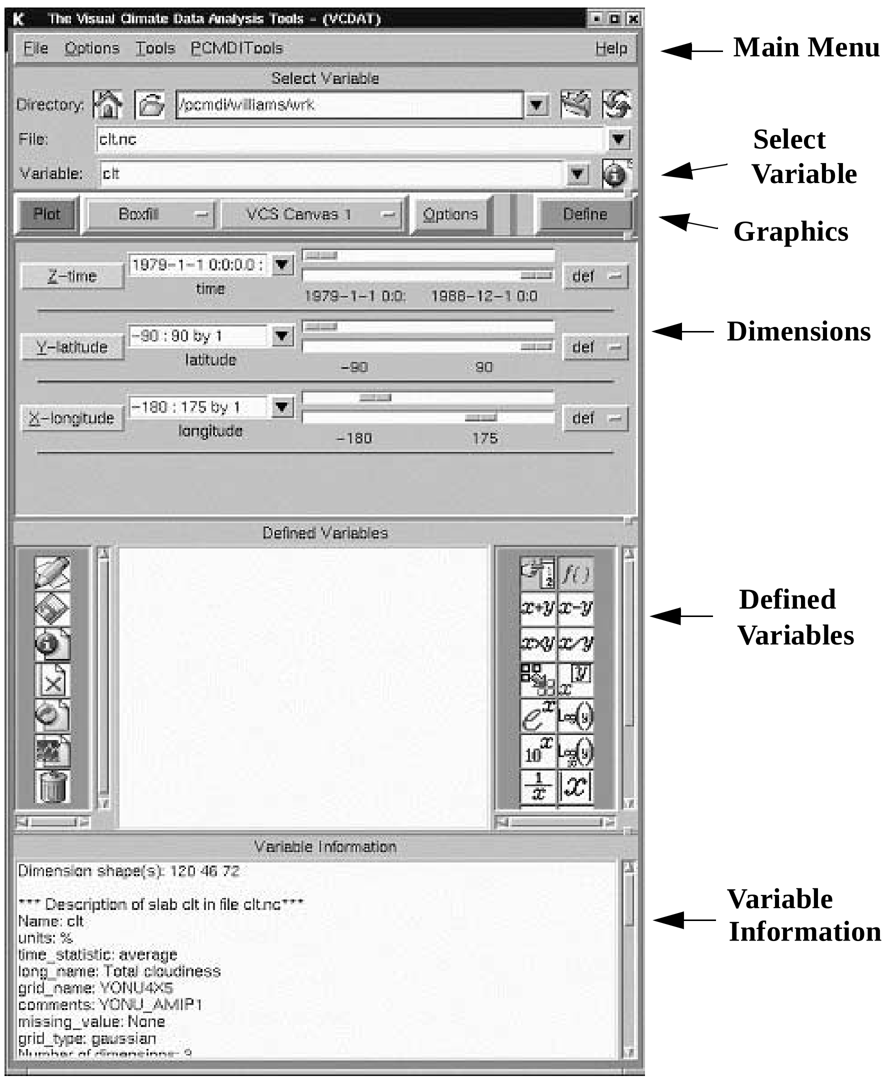
Example 4: To define a variable in memory:

1. Select the directory where the data is stored,
2. Select the desired file that is located in the directory,
3. Select a variable from the file,
4. If desired, select dimension sub ranges.
5. Select the "Define" button,
6. Select the defined variable located in the "Defined Variables" window located just below the "Dimension" panel,
7. Click the "Plot" button.

2.1.1 Description of the VCDAT Interface

There are six sections of the VCDAT interface (figure 1) that are listed here from top-to-bottom: Main Menu (i.e., File, Options, Tools, PCMDI Tools, Help); Select Variable (Directory, File, and Variable); Graphics (i.e., Plot, Boxfill, VCS Canvas 1, Options, Define); Dimensions (Z-time, Y-latitude, X-longitude); Defined Variables (Operational Icons, Defined Variable List, Function Icons); and Variable Information.

Figure 1. The VCDAT Interface



2.1.1.1 Main Menu (i.e., File, Options, Tools, PCMDI Tools, Help)

There are five static pull down options in VCDAT's main menu. In addition, the user can add their own customized pull down menus to VCDAT's main menu.

- **File**: Under the "File" option, the user can choose to: open data files located on their local disk, select searches for specific data types (e.g., netCDF, XML, etc.), send plots directly to local printers, save the state of the system for total recall, read Python or VCS script files into the system, and exiting out of VCDAT.
- **Options**: The "Options" option sets information referring to a selected or defined variable, or sets the state of the graphical user interface. For example, the "Select Predefined Region" selects a region of the globe (e.g., "Africa", "Asia", etc.). Thus, every selected or defined variable will automatically be set to the predefined region. Note: This works only if the x-axis equals longitude and the y-axis equals latitude.

  Dimension aliases for longitude, latitude, time, and level can also be set under the "Options" pull down menu. For example, the user can set "X", "lon" and "X_Long" as aliases for longitude. So when VCDAT sees these dimension names it will interpret them as longitude axes.

  If the user modifies the VCDAT GUI and wants to save the settings for the next session, then select "Save GUI State…" located under this option.
- **Tools (IDLE)**: The "Tools" option displays additional GUIs that allow the user to: create new script files for edit, edit old script files, issue keystroke commands, add new user menu options to the "Main Menu" (note, the new menu option will be placed between the PCMDITools and Help menu options), and view recorded script commands.

When invoked, the "Command Line Window…" option is really executing the Integrated DeveLopment Environment (IDLE) developed by Guido van Rossum, who is also the developer of Python. The Python Shell Window gives the user access into Python's interactive mode. This Python Shell Window has been slightly modified to allow the user to register keystroke script commands and GUI button commands simultaneously and seamlessly in VCDAT.

When the "Python Shell Window" (i.e., called "VCDAT's Command Line Window" in VCDAT) appears, the user should read the information describing hooks into VCS. For additional online information about the "Python Shell Window", select the "F1" button while running it.

The script editors are also part of IDLE. IDLE can also be invoked outside of VCDAT. That is, at the prompt type, "idle". For more information on IDLE, visit: http://www.python.org/idle.

To add more menus to "Main Menu", the user must select the "Add User New Menu…" menu item. See help balloons to assist with the addition of new menus. Once the new menu has been added, the user must select "Save GUI State…" located under "Main Menu" -> "Options" to retain the new menu for later VCDAT sessions.

- **PCMDITools**: The "PCMDITools" option contains the start of time tools and statistical functions deemed necessary (by the PCMDI climate researchers) to do common fundamental analysis. For the "Time Tools", the data must have a time dimension in order to work. Under "Options", see "Dimension Aliases for" specifying additional time axis names.

  For in-depth information on this tools and their functionality, see the cdat_utilities.pdf document.
- **Help**: The "Help" option allows the user to turn the help balloons off or on, display CDAT related web sites by means of the user's client web browser, and displays information about VCDAT to the user (for example, version number and place of execution).

2.1.1.2 Select Variable

The main purpose of this section is to select a variable for analysis and/or display. There are five main components here: the "Directory"; the "Database", this component take the place of the "Directory" when it is invoked; the "File"; the "Datasets", this component takes the place of the "File" when the "Database" is invoked, and the Variable.

- **Directory**: The "Directory" component of "Select Variable" contains: a home directory icon, which if selected will take the user directly to their $HOME directory; a file selection browser icon that will display a file selection GUI for selecting the directory and a file; an input window for specifying the directory by means of a keypad; a bookmark icon for saving favorite working directories; and a cycle icon for cycling through the list of favorite working directories.

  Note: the "Directory" input window will turn salmon pink when the user enters text. The textual information requesting a new directory will not be excepted until the <Return> key is depressed.
- **Database**: The "Database" component is displayed only when the user has changed the choice button from "Directory" to "Database". If the user has not installed "esg" on their system and the environment variable "CDMSROOT" has not been set, then the "Database" option will not appear. If the "Database" option is selected, then a popup will appear allowing the user to log onto a remote database. After connecting to a remote database the "File" component will change to display "Datasets".
- **File**: The "File" component displays all the files located in the directory. To view only a specific file type, select under "Main Menu" -> "File" -> "Open File Types" the desired file specifications.

- **Datasets**: The "Datasets" components displays all the datasets located in the database. This is only visible if the "Database" component is visible.
- **Variable**: The "Variable" component displays the variables, axes, and bounds in a file or dataset. To alter the viewing of "Variable" (i.e., don't display or display axes, bounds and weights), toggle "Main Menu" -> "Options" -> "View axes in Variable List" and "Main Menu" -> "Options" -> "View bounds and weights in Variable List".

2.1.1.3 Graphics (i.e., Plot, Boxfill, VCS Canvas 1, Options, Define)

There are only four components to "Graphics", but because of screen real estate and design considerations, the "Define" button was added to this row. Therefore, "Graphics" consist of five components: the "Plot" button, the "Graphics Method" (e.g., "Boxfill", "Isofill", etc.) choice button, the "VCS Canvas" choice button, the graphics "Options" menu, and the "Define" button.

- **Plot**: After the "Variable" has been selected, the user can select the "Plot" button to display the variable. The user can also select the "Plot" button to display a "Define Variable", which is described later in this document.
- **Graphics Method**: By default, the "Boxfill" graphics method will be displayed. To choose a different graphics method, the user must depress and hold the choice menu button. Then move the pointer over the desired graphics method (e.g., "Isofill"). At this point, the user can reselect the "Plot" button to redisplay the variable.
- **VCS Canvas**: VCDAT allows up to four VCS Canvases. More can be added if necessary. The user can select which canvas to display the variable information. Note, if the user wishes to display the information on the "Background Canvas", then rendering is done in memory and no canvas drawing is visible to the user, but the

user can still send the unseen plot to the printer or to a graphics output file. See "Main Menu" -> "Save Plot As" and "Main Menu" -> "Print Plot On" for sections on file output and printer selections. Also, view the VCS document for details on the PCMDI_GRAPHICS directory and the HARD_COPY file.

- **Options**: The "Options" menu displays additional GUIs for controlling the outcome of the plot. These plot options are: "Continents Types", "Page Orientation", "Overlay", "Isoline Labels", "Annotation", "Set VCS Canvas Geometry", "Set Min Max Values", "Set Graphics Method Attributes", Number of Plots on VCS Canvas", "Set Plot Map Projection", "Editors", "Animate", "Clear VCS Canvas", and "Close VCS Canvas".
- **Define**: After the "Variable" has been selected, the user can select the "Define" button to transfer the variable from a file or remote database to memory. The selected variable will be visible in the "Defined Variables" below.

  The "Define" button is also used to save a defined variable under a new name and can be used to overwrite an existing defined variable. That is, in the "Defined Variable List" window, select a defined variable, and then select the "Define" button. A pop up will appear with simple instructions for both actions.

2.1.1.4 Dimensions Panel

The "Dimension Panel" is blank if no variable or defined variable is selected. It is located just under the "Graphics" section. It can currently display up to five dimensions. But more display dimensions can be added if necessary. The displayed dimensions are separated by lines and contain four components: the "Dimension Menu"; the "Dimension Selection"; the "Dimension Sliders"; and the "Dimension Functions".

- **Dimension Menu**: The "Dimension Menu" allows the user to change the view of the "Dimension Selection" from axes defined values to index values. For example, from longitude or latitude coordinate values to index values. In the case of the time dimension, the user can change the time coordinate values, i.e., "units since base time" (See "Handling Time" on page 41.) to either raw time values or index values.

The user can get the weights of an axis' by selecting the "Get Axis Weight Values" menu item. The weights variable will be stored in memory and displayed in the "Defined Variable" window below.

By selecting the "Replace Axis Values", the user can replace the axis node values with a defined variable, provided that the axes and the defined variable are the same size. The new axis must be visible in the "Defined Variables" window.

To reorder the dimensions, depress and hold the "Re-Order Dimensions". Then select the desired dimension to trade places. The "Dimensions Panel" will restructure itself accordingly.

- **Dimension Selection**: The "Dimension Selection" allows the user to select a single node value, a subset of node values, and a selection of every nth node value. When selecting the arrow button to the right will display the list axes node values. The highlighted value indicates the selected node(s). The input text window representation the dimension as "first: last by stride".

  The user can enter the node value(s) in the input text window. When entering values, the input window becomes salmon pink. To except the changes, the user must depress the <Return> key. To select every nth node value, the user must enter the stride number after the "by" and then depress the <Return> key.

  To select a single node, the user can either enter the value in the input text window or select the arrow button to the right and then select the desired node value. To select a sub-range, the user can either enter the values in the input text window or select the arrow button to the right and then select the second desired node value.

- **Dimension Slider**: The slider bars are yet another way to select the first and last values of a sub-range or single node point. The top slider changes the first selected node value and the bottom slider changes the last selected node value. The values below the second slider bar represent the first selected node value and the last selected node value.

- **Dimension Functions**: Each dimension can have a function operate on it exclusively. Select the choice menu button for a detailed description of each function operation. The last two functions "awt" and "gtm" are slightly different. "awt" allows the user to replace the weights before operating the weighted average function. The new weights must be located in the "Defined Variables" window before using this function. The geometrical mean "gtm" is generated by the following function: gtm(x) = exp(mean (log(x)))).

2.1.1.5 Defined Variables

The "Defined Variables" section operates on variables that are stored in memory. Variables can be defined in memory by selecting the "Define" button located in the "Graphics" section. Variables can also be defined by operations located in the "Main Menu" section. For example: selecting from "Main Menu" -> "PCMDITools" -> "Statistics" -> "Mean" on a variable would produce "_Statistics_mean_variablename ()" in the "Defined Variables List". Other possible ways to produce a defined variable are: by the use of "Function Icons" and command line operations via VCDAT's "Command Line Window". For example: typing "import MV; a = MV.array((1,2,3))" in the VCDAT "Command Line Window" would produce "a (3)" in the "Defined Variables List".

Note that everything associated with defined variables has the same background color as the "Define" button located in the "Graphics" section.

- **Operational Icons**: The first column of icons (from top to bottom) allows the user to: edit the variable's attributes, save the variable to a netCDF file, display variable information, move selected defined variables to the trashcan, move all variables to the trashcan, view logged information about the size of variables in memory, and permanently remove or restore defined variables.

- **Defined Variables List**: The "Defined Variables List" is located to the right of the "Operational Icons" and contains, in alphabetical, the list of all variables stored in memory. As the user selects variables in the list, a record of the order of selection is kept. Among other things, this selection list is important when it becomes necessary to plot or do calculations by means of the "Function Icons".
- **Function Icons**: The "Function Icons" are located to the right of the "Defined Variables List". There are two modes for calculating defined variables. If the user moves the pointer over the top left icon, a detailed description of how the calculation works in both modes is given.

2.1.1.6 Variable Information

The "Variable Information" section immediately displays variable or defined variable information to the user. To quickly browser through many variables and examine their content, resize the variable information panel by selecting the small button located to the upper right of the "Variable Information" text. While keeping the button depressed, move the panel to the desired location.

2.2 A brief introduction to Python scripting.

Python is documented down to the last detail at **http://python.org** and the SourceForge website **http://python.sourceforge.net**. There are now many books about it. We can recommend the free tutorial available at www.python.org in the documentation section. The book “Learning Python” from O’Reilly Press is another excellent resource. This section provides a very brief introduction to some basic concepts aimed at “getting started”. The interested reader can refer to any of the other sources referred to above for details.

We begin with the traditional “Hello World” program. To start the python interpreter type:

```
% python

Python 2.2.1c2 (#1, Apr 11 2002, 12:36:10)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)]
      on linux2
Type "help", "copyright", "credits" or "license" for
      more information.
>>> print "Hello World"
Hello World
```

Programs can be placed in files with a .py extension such as:

```
# helloworld.py
print “Hello World”
```

The program can then be executed by providing the name of the program file to the interpreter as follows:

```
% python helloworld.py
Hello World
%
```

2.2.1 Variables and arithmetic expressions:

A simple program can best illustrate the use of arithmetic expressions.

```
# simple_program.py
```

```
pi = 3.1415
degrees = 0
while degs <= 90:
    rads = degs * pi / 180.0
    print degs, " degrees = ", rads, " radians"
    degs = degs + 10.
```

The output of this program is the following table:

```
0  degrees =  0.0  radians
10.0  degrees =  0.174527777778  radians
20.0  degrees =  0.349055555556  radians
30.0  degrees =  0.523583333333  radians
40.0  degrees =  0.698111111111  radians
50.0  degrees =  0.872638888889  radians
60.0  degrees =  1.04716666667  radians
70.0  degrees =  1.22169444444  radians
80.0  degrees =  1.39622222222  radians
90.0  degrees =  1.57075  radians
```

Python is a dynamically typed language (i.e you do not need to declare the variable types *apriori*) and the names can represent different types depending on the arithmetic operations performed. In the above example, "degs" was initially set to the integer 0, and is seen in the first line printed. Subsequently a real number (10.) is added at each stage of the loop and this changed the type of "degs" as it can be seen that "degs" values printed are real. Note that the above example is contrived to illustrate this point and there is no real reason to add a real value 10.0 in the loop.

The output of the program looks less than ideally formatted. To make it look better, we can make use of format strings. For example:

```
>>> print "%3d degrees = %0.4f radians" %(degs, rads)
```

would produce output that looks like this:

```
  0 degrees = 0.0000 radians
 10 degrees = 0.1745 radians
 20 degrees = 0.3491 radians
```

```
 30 degrees = 0.5236 radians
 40 degrees = 0.6981 radians
 50 degrees = 0.8726 radians
 60 degrees = 1.0472 radians
 70 degrees = 1.2217 radians
 80 degrees = 1.3962 radians
 90 degrees = 1.5708 radians
```

The format strings **`%d`**, **`%s`** and **`%f`** denote integers, strings and floats respectively.

2.2.2 Conditional statements:

The `if` and `else` statements provide an easy way to perform tests. For instance:

```
>>> if x != y:
>>>     print ‘x is not equal to y’
>>> else:
>>>     print ‘x and y are equal’
```

The indentation is required to isolate the **`if`** and **`else`** clauses, but the else clause is optional. Do nothing clauses can be created by using the **`pass`** statement.

```
>>> if x != y:
>>>     pass
>>> else:
>>>     print ‘Voila’
```

Multiple test cases can be implemented using the **`elif`** clause.

```
>>> if x == ‘n’:
>>>     print ‘Answered no’
>>> elif x == ‘y’:
>>>     print ‘Answered yes’
>>> else:
>>>     print ‘invalid answer’
```

Boolean expressions can be formed by using **or**, **and**, and **not** keywords.

```
>>> if x>y and z > x:
>>>     print 'z is the max value'
>>> if not (x==z or y==z or x==y):
>>>     print 'There are no equal values'
```

2.2.3 File Input and Output:

To open a text file and read its contents you would

```
>>> f = open("myfile.txt")
# The above line returns a file object
>>> line = f.readline()
# The readline() method is invoked on file
# and one line is read from the file.
# The following section keeps printing and reading
# subsequent lines of data while there are new lines to
# be read
>>> while line:
>>>     print line
>>>     line = f.readline()
# The while loop is exited when there are no more lines
# to be read. To close the open file:
>>> f.close()
```

2.2.4 Lists and Tuples:

Lists and tuples are sequences of arbitrary objects. Lists can be created by:

```
>>> mylist = ["a", 1.0, "c", 4]
```

Note that you can mix items of any type in a list. You can also have lists nested inside lists.

```
>>> my_other_list = [“a”, “b”, [1,2,3], “d”]
```

The lists are indexed by integers starting with zero. To see the first item in mylist, you would:

```
>>> firstitem = mylist[0]
# returns the item “a”
# To set an item:
>>> mylist[2] = “x”
# Changes the item in the index position 2.
>>> print mylist
[“a”, 1.0, “x”, 4]
```

To append new members to the list, the append() method is used as follows:

```
>>> mylist.append(“nextitem”)
>>> print mylist
[“a”, 1.0, “x”, 4, “nextitem”]
```

Similarly, individual items can be removed from the list using the remove() method:

```
>>> mylist.remove(1.0)
>>> print mylist
[“a”, “x”, 4, “nextitem”]
```

You can also insert items into specific positions:

```
>>> mylist.insert(1, “inserted_item”)
>>> print mylist
[“a”, “inserted_item”, “x”, 4, “nextitem”]
```

Lists can be concatenated by using the “+” operator:

```
>>> newlist = mylist + [9, 10, 11]
>>> print newlist
[“a”, “inserted_item”, “x”, 4, “nextitem”, 9, 10, 11]
```

Tuples are very similar to lists and are constructed by using parentheses instead of brackets or just a comma-separated list.

```
>>> mytuple = (1, 2, -3)
>>> myvector = (magnitude, direction)
# is the same as
>>> myvector = magnitude, direction
```

Tuples support the same operations as are supported by lists except the appending or modification of elements after they are created.

2.2.5 Loops:

We saw the simple loop using the **`while`** statement in the previous examples. Other looping constructs such as the **`for`** statement are available to the user. An example of its usage is:

```
>>> for i in range(3):
>>>     print "10 raised to ", i, " is ", 10**i
# will produce the result
10 raised to 0 is 1
10 raised to 1 is 10
10 raised to 2 is 100
```

Note that a=range(3) is equivalent to a=[0,1,2]. Similarly the range(1, 6) is equivalent to [1,2,3,4,5] and range(10, 8, -1) is equivalent to [10, 9]. To generalize, range(i, j, k) produces a list of integers from i to j-1 using a stride k. If i is omitted, it is taken to be zero and k defaults to 1 if omitted. A more efficient (in terms of memory and runtime) is the **`xrange()`** function used exactly like **`range()`**.

One can also loop through lists using the **`for`** statement so:

```
>>> for item in mylist:
>>>     print item
```

2.2.6 Dictionaries:

A dictionary allows you to associate values index by keys. Dictionaries can be created using values in curly braces like this:

```
>>> a = {
         “name” : “CCM3”,
         “center” : “NCAR,”
         “model_of” : “Atmosphere”
        }
```

To access members of the dictionary, we use the key-indexing facility

```
>>> model_name = a[“model”]
>>> modelling_center = a[“center”]
```

The keys associated with a dictionary can be obtained as a list by:

```
>>> b = a.keys()
```

and you can check the dictionary membership by using the has_key() method:

```
>>> if a.has_key(“model_of”):
>>>     print a[“model_of”]
>>> else:
>>>     print “Unknown model of”
```

2.2.7 Functions:

A function can be created using the `def` statement as shown below.

```
def myadd(a, b):
    c = a*10 + b
    return c
```

To invoke the function we do the following:

```
>>> addvalue = myadd(2, 5)
```

It is possible to return multiple values using comma separated names in the return statement inside the function.

```
def myaddsub(a, b):
    c = a*10 + b
    d = a*10 - b
    return c, d
```

In this case the function is invoked as follows:

```
>>> addvalue, subvalue = myaddsub(2, 5)
```

The function definition can also be done in a way such that default values for input parameters can be set.

```
def myaddsub(a, b, base=10):
    c = a*base + b
    d = a*base - b
```

Then, when you need base to take a different value than the default 10, you can

```
>>> addvalue, subvalue = myaddsub(2, 5, base=2)
```

2.2.8 Modules:

To keep your programs manageable as they grow in size, you may want to break them up into several files. Python allows you to put multiple function definitions into a file and use them as a module that can be imported into other scripts and programs. These files must have a .py extension. For example

```
# file my_function.py
def minmax(a,b):
    if a <= b:
        min, max = a, b
    else:
        min, max = b, a
    return min, max
```

To use the above module in other programs, you would use the `import` statement.

```
>>> import my_function
>>> x,y = my_function.minmax(25, 6.3)
```

2.3 How to get the tutorials and data

The tutorials are designed to introduce you to the most common operations for climate data analysis. They are available as a Python script from our download area under “Tutorials”. If you want to try executing these examples, follow these steps:

1. From the Tutorials section of the download facility at **http://cdat.sf.net**, download the data files tarball (**cdat_tutorial_data.tar.gz**) and unpack it.
2. Download the tutorial files tarball (**cdat_tutorials.tar.gz**) and unpack it. In each tutorial you may need to edit the line(s) near the top that sets the location of the data files to match the location where the data will be on your system.
3. At some point during the tutorials you may wish to also read “Reading, creating, and altering variables” on page 33 for a discussion of the difference between several array-like abstractions that our software uses: numerical arrays, masked arrays, transient variables, and file variables.

2.4 What do the tutorials cover?

There are five tutorial files provided with data required to run them. The tutorials are CDAT scripts that give the user a flavor of how scripts work and to give an idea of how to accomplish specific tasks. A brief description of the tutorials follows:

2.4.1 getting_started_tutorial.py

This tutorial is the first one to study. The tutorial consists of three examples:

2.4.1.1 Example 0: The basics

This example runs through some basic operations such as opening, closing, writing to files, and reading data by location or at specified time intervals. The tutorial is presented with a series of questions and answers in the style of a FAQ.

2.4.1.2 Example 1: Dealing with data from other sources

This example shows (in excruciating detail) how to read in Fortran formatted data, use it within CDAT, and writing to a netCDF file. Some simple averaging operations are carried out on the data. This example also shows how to change/insert metadata for the variable.

2.4.1.3 Example 2. Using Masks.

This example shows how to generate data masks, applying masks and averaging using area weights.

2.4.2 times_tutorial.py

This tutorial demonstrates the uses of the time averaging functions. Basic examples cover topics such as computing the December-February seasonal means, computing the climatology, anomalies (departures) from climatology, construction of seasons not already defined, customization, and use of powerful criteria to specify minimum temporal coverage and data distribution.

2.4.3 statistics_tutorial.py

This tutorial covers some of the basic statistical functions such as rms, correlation, mean absolute difference and their usage with climate data.

2.4.4 vcs_tutorial.py

This tutorial guides you through some basic plotting functions and features to visualize the data and produce presentation quality output. This is by no means an exhaustive demonstration of features - just a very basic set of capabilities are addressed. Specific examples cover creating and altering “isofill” graphics methods, creating and altering display templates, producing output files for printing and displaying and changing colormaps etc.

2.4.5 xmgrace_tutorial.py

In this tutorial, the interface to the XmGrace utility is demonstrated by showing simple plot generation and customization. XmGrace is a plotting tool developed independent of CDAT and has a wide user base. To download and install XmGrace, see **http://plasma-gate.weizmann.ac.il/Grace**

2.5 Documentation and Help

The **CDAT home page http://cdat.sf.net** is your source for documentation- both online and printable and support. Documentation is available to help users at various levels of expertise starting with the beginning user’s guide (this document) to the advanced user who needs a quick reference to commands or details of usage in the programming API reference guides. CDAT makes heavy use of Numerical Python, a fast array facility for Python. The documentation for Numerical Python is at **http://numpy.sourceforge.net**.

2.5.1 Online Documentation

2.5.1.1 Using VCDAT

The VCDAT graphical user interface has a “help” button that provides access to all the documentation that a user may require at the click of a button. The interface also has helpful information in pop-up balloons that instruct the user when the cursor is moved over the buttons.

2.5.1.2 Using “docstrings”

Python also has a powerful feature: most objects in Python, including the modules, classes, and functions, have documentation strings (“docstrings”) attached to them. The special attribute name “__doc__” (which has two underscores on each end) is used to access this string supplied by the module author. If you have an object x, try the command

```
>>> print x.__doc__
```

2.5.1.3 Using pydoc to generate documentation

You can also use the pydoc utility. This is a standard facility for extracting documentation from Python installations.

You can generate Python documentation in HTML or text for interactive use.

2.5.1.4 Interactive use

In the Python interpreter, do "from pydoc import help" to provide online help. Calling help(thing) on a Python object documents the object. Also, typing help() will put the user in a help environment indicated by the prompt changing to

```
help>
```

2.5.1.5 From the shell

At the shell command line outside of Python: Run "pydoc <name>" to show documentation on something. <name> may be the name of a function, module, package, or a dotted reference to a class or function within a module or module in a package. If the argument contains a path segment delimiter (e.g. slash on Unix, backslash on Windows) it is treated as the path to a Python source file. Run "pydoc -k <keyword>" to search for a keyword in the synopsis lines of all available modules.

2.5.1.6 Starting a browser / server

pydoc -g starts an HTTP server and also pops up a little window for controlling it.

2.5.1.7 Writing out HTML

Run "pydoc -w <name>" to write out the HTML documentation for a module to a file named "<name>.html".

2.5.1.8 Using happydoc.

A utility named “happydoc” is able to generate documentation for modules written in Python.

2.5.2 Printable documentation:

2.5.2.1 Quick Reference

The documents *cdms_quick_start.pdf* and *vcs_quick_start.pdf* provide a quick reference of useful commands in the core CDAT modules; **cdms** and **vcs** respectively on a single page that is useful even to the experienced user.

2.5.2.2 Programming reference guides.

The core CDAT modules **cdms** and **vcs** are extensively documented in separate documents that can be downloaded from **http://cdat.sf.net**. The most up-to-date versions of the following guides can be found there.

- **cdat.pdf**: This document
- **cdms.pdf**: C*limate Data Management System*
- **vcs.pdf**: *Visualization and Control System: Python Command Line and Application Programming Interface*
- **cdat_utilities.pdf**: *CDAT Utilities Reference Guide*
- **numpy.pdf**: *Numerical Python*

2.5.3 Getting Support

The **CDAT Home Page** is cdat.sf.net. CDAT is hosted at SourceForge, a free service provided by VA Linux, Inc. to the Open Source Community. SourceForge enables us to provide the following services for users:

- A release facility, where users can download binary and source releases and see release notes.
- A bug-tracking facility, where users can submit bugs and track their status, and receive mail when they are fixed.
- A facility to request feature enhancements.
- A mailing list for discussion of CDAT (cdat-discussion@lists.sf.net).

You can use these facilities without registering at SourceForge, but registration, which is quick, easy, and free, will enable you to participate in the fullest possible way. In particular, it is very helpful to us if you are registered when you submit a bug report.

CHAPTER 3

What's in CDAT

3.1 File/Data Handling

3.1.1 File I/O

3.1.1.1 Reading/writing data from/to self-describing formats

A variable (defined in more detail in "Reading, creating, and altering variables" on page 33) can be obtained from a file or collection of files, or can be generated as the result of a computation. Files can be in any of the self-describing formats netCDF, HDF, GrADS/GRIB (GRIB with a GrADS control file) or xml files generated by CDMS.

For instance, to read the variable **u** from file **sample.nc**:

```
>>> import cdms
>>> f = cdms.open('sample.nc')
>>> u = f('u')
```

Data can be read by index or by world coordinate values. The following reads the n-th timepoint of **u**:

```
>>> u0 = f('u',time=slice(n,n+1))
```

and this reads **u** at time 366.0:

```
>>> u1 = f('u',time=366.)
```

A variable can be written to a file with the **write** function:

```
>>> g = cdms.open('sample2.nc','w')
>>> g.write(u)
<Variable: u, file: sample2.nc, shape: (1, 16, 32)>
>>> g.close()
```

More details on reading and writing data can be found in *Climate Data Management System* (**cdms.pdf**).

3.1.1.2 Reading/Writing Fortran formatted data. (Scientific.IO)

The Scientific Python package maintained by Konrad Hinsen contains numerous subpackages useful in scientific applications. The "IO" subpackage is useful for reading and writing data using Fortran format statements. The example shown below demonstrates how easy it is to read Fortran formatted data.

```
>>> f = open(ascii_filename, 'r')
# Import the module that does the work.
>>> from Scientific.IO import FortranFormat
# Declare the fortran formats used to create the data.
>>> ff1 = FortranFormat.FortranFormat('2i6')
>>> ff2 = FortranFormat.FortranFormat('12i6')
>>> data_line = f.readline()
>>> mon, yr = FortranFormat.FortranLine(data_line, ff1)
# Now define an array to read the data into.
>>> import Numeric
>>> T_array = Numeric.zeros((14,))
# In the next line you are assigning the values.
>>> T_array[start_index: end_index] = \
      FortranFormat.FortranLine(f.readline(), ff2)
```

```
# Note: You must have previously defined T_array.
#See tutorial examples for more details.
```

3.1.1.3 Reading data from ASCII text files. (asciidata)

The **asciidata** module is useful in reading (and writing) data that is in ASCII format with the ability to specify tab or comma or space delimited fields. This is particularly useful to deal with importing (or exporting) spreadsheet data. For example,

```
>>> import asciidata
>> time, pressure =
      asciidata.comma_seperated('myfile.txt')
```

The IO subpackage of Scientific also contains other useful facilities of this type.

3.1.1.4 Reading/Writing unformatted data (binaryio)

To handle binary or unformatted data, the module **binaryio** offers a convenient interface. The following example illustrates the use of this module. Note that binary files are platform and compiler dependent.

```
# To write binary data.
>>> import binaryio
>>> iunit = binaryio.bincreate('filename')
>>> binaryio.binwrite(iunit, \
      my_array_with_upto_4_dimensions)
>>> binaryio.binclose(iunit)
# To read binary data
>>> iunit = binaryio.binopen('filename')
>>> y = binaryio.binread(iunit, n, ...)
>>> binaryio.binclose(iunit)
```

3.1.2 Reading, creating, and altering variables

Climate data comes in many different file formats, organized in many different ways. The **cdms** package gives you a uniform interface so that you can write processing algorithms and graphics that will work with a wide variety of different data formats.

In CDMS, the basic unit of data is the ***variable***. A ***variable*** is essentially a multidimensional array, augmented with a ***domain*** and with ***metadata***. The domain describes the spatial location and temporal information associated with the array. For example, a variable on an orthogonal grid may have a domain consisting of (but not limited to) time, level, latitude, and longitude axes. The metadata associated with a variable consists of a collection of *attribute-value* pairs. The set of attributes of a variable is not predefined, although some attributes, such as the units and the missing data value, are commonly used.

A variable may be stored in a single physical file or in a collection of files, called a ***dataset***. The files themselves may be in any of several self-describing formats, such as netCDF, HDF, and GraDS/GRIB. CDMS provides a uniform interface to data, so that the same processing algorithm or graphics routine will work with any of the supported data formats.

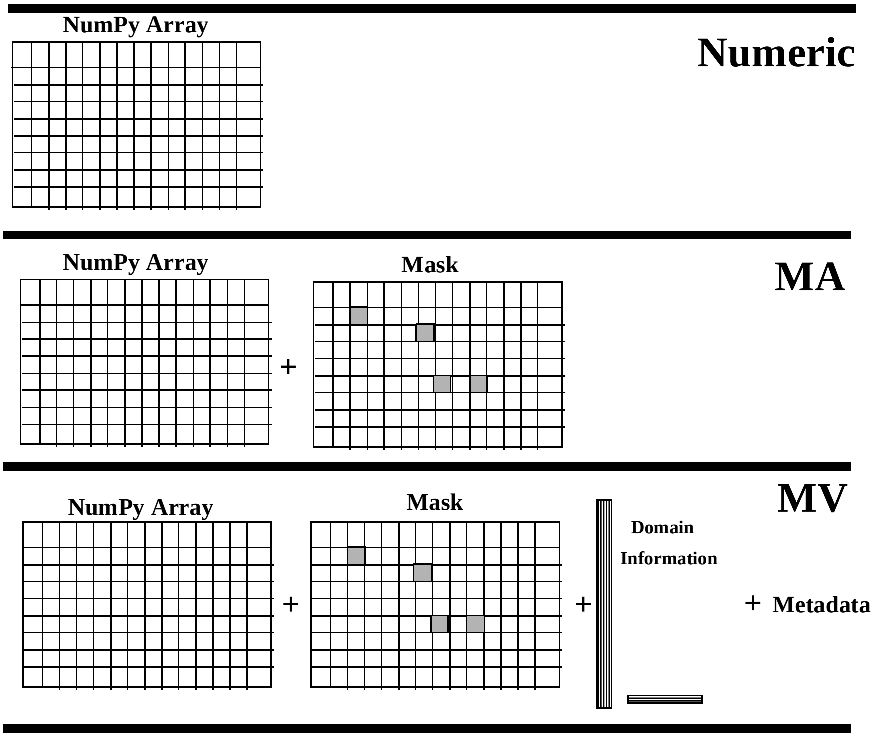
For computing tasks, variables can be used much like arrays. The common arithmetic functions are defined for variables, as well as I/O and slicing operations. One advantage of using CDMS variables for computation is that the associated domain and metadata information is carried along with the computation. The benefit of this approach is that, for example, if we average over time, and then plot the result, the plotting routines can still be aware that the other dimensions represent latitude and longitude and draw the continental outlines, do projections correctly, etc. If the result were merely the averaged data, that interpretation of it would have been lost. This frequently results in much simpler scripts than otherwise would be the case.

However, CDAT is also designed for extensibility. The capability to add external C or Fortran modules is extremely important for advanced applications. For example, many external modules use the "array" type defined by the Numerical Python (NumPy) module. To accomodate the need for interoperability, CDAT provides a hierarchy of representations for data arrays:

- Numeric Array: a multidimensional array, all elements have the same datatype (real, integer etc.). This type is supported by the Numerical Python (NumPy) module.
- Masked Array (MA): A Numeric array with an optional missing data mask. Operations on these compute the mask of the result.
- Masked Variable (MV): A masked array having a domain and metadata. Computations carry along the domain and metadata information where possible. A masked variable in memory is referred to as ***transient variable*** and a masked variable in a dataset is called a ***file variable***

A schematic of the array hierarchy is shown in Fig. 1.

FIGURE 1. **Relationships between array abstractions.**



While the above figure represents the general schematic of the different array constructs, the user should be aware that the individual constructs also include functions (for e.g. arithmetic functions) that operate on these arrays. Some of these functions are intended to convert arrays of one type to the other. These allow the user to convert the array to the necessary level to suit the algorithm. The conversion from one level of array abstraction to the other is dealt with in the following sections.

3.1.2.1 Constructing Numeric arrays

As explained in the Numerical Python manual, a Numeric array can be constructed in many ways. The basic usage is to apply the array “constructor” Numeric.array:

```
>>> import Numeric
>>> x = Numeric.array (s)
```

where s can be a Python list, tuple, or another Numeric array.

For many applications, space allocation for newly defined variables needs to be controlled so as not to run out of memory. CDAT allows for some control over this. For example, ff you do not require a separate copy of the data, copy=0 can be added.

```
>>> x = Numeric.array(s, copy=0)
```

If you want to control the type of the data, you can supply a typecode, usually in the form of one of the abstract constants supplied in Numeric for this purpose. For example,

```
>>> y = Numeric.array([1,2,3], Numeric.Float)
```

would create y as an array containing the floating-point numbers 1., 2., and 3. By the way, a very frequent beginner error is to say something like:

```
>>> y = Numeric.array(1,2,3)  # Error !
```

which is an error that will result in a very strange message:

```
>>> import Numeric
>>> y = Numeric.array(1,2,3)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    y = Numeric.array(1,2,3)
TypeError: typecode argument must be a string.
>>>
```

What has happened is that the second argument to Numeric.array was the number 2, and Numeric is expecting one of its typecode letters such as 'd' in that position.

3.1.2.2 Numeric to MA

Given a Numeric array x, if we wish to construct a masked array, it will be one of these cases:

1. We want to treat x as a masked array xm, but none of its values are currently missing, and wish to share x's data space, so that a modification to xm is also a modification to x. Solution:

```
>>> import MA
>>> xm = MA.masked_array (x)
```

2. We want to treat x as a masked array xm with no values considered missing and without sharing x's space. Solution:

```
>>> xm = MA.array (x)
```

3. We want to treat x as a masked array xm with a certain value v treated as a missing value. Solution:

```
>>> xm = MA.masked_value (x, v).
```

   See the MA manual for other arguments, such as controlling the precision with which a value in x must be equal to v in order to be considered missing.

4. We want to treat x as a masked array xm but with those values considered missing that correspond to non-zero values in an array m of zeros and ones. Solution:

```
>>> xm = MA.array(x, mask=m).
```

   Again, see the manual for more options on this.

5. The array x is of a numeric type and we want to mask all those values greater than a certain value v. Solution:

```
>>> xm = MA.masked_greater (x, v).
```

   Similarly, there are functions masked_greater_equal, masked_less, masked_less_equal, masked_equal.

3.1.2.3 Numeric or MA to Transient Variable (MV)

The array constructor cdms.MV.array is similar to the MA.array constructor but allows additional arguments specifying the metadata. For full usage, see *Climate Data Management Systam* (**cdms.pdf**). The options discussed in the previous section will produce transient variables as long as you use the functions in cdms.MV instead of those in MA.

The most frequent additional argument to MV.array is to specify a list of axes using the axes=alist argument. Often these axes have been extracted from another variable using getAxis or getAxisList, or created from data.

For example, if you were going to operate on variable **v** using some process that returns a Numeric array, and in the process remove its time dimension, you might do something like this:

```
>>> import cdms, MV
>>> f = cdms.open('../data_directory/clt.nc')
>>> v = f('clt')
>>> alist = v.getAxisList(omit='time')
>>> u = v(time=(‘1979-1-1’, ‘1979-1-1’), squeeze=1)
>>> x = MV.array(u, axes=alist)
```

Now, x is again a transient variable and it has the geographic information reattached so that plots will show the continents. Note that it is usually not necessary to do this. Any ordinary arithmetic, and any use of the functions in MV, will return transient variables with appropriate axes.

As mentioned above, transient variables can have associated *attribute*s which are accessed and set using the Python dot notation:

```
>>> u.units=’m/s’
>>> print u.units
m/s
```

Attribute values can be strings, scalars, or 1-D Numeric arrays. More details on variables and the available functions and methods can be found in *Climate Data Management System* (**cdms.pdf**).

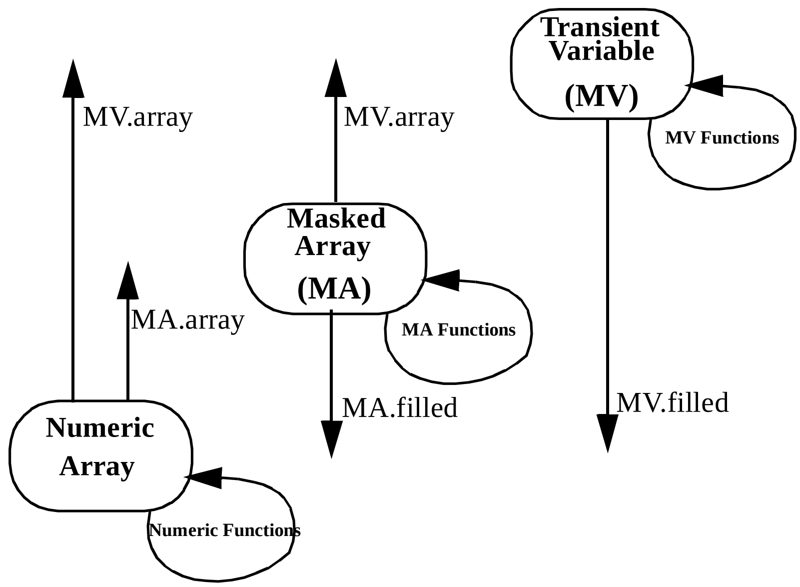
3.1.2.4 MA or Transient Variable to Numeric

Many packages that support Numerical Python arrays have been developed. You can find links to some of them on the CDAT website. These packages work with Numerical Python arrays. For these and other applications (such as modules that are written in Fortran that expect data as Numeric arrays), it is necessary to convert existing arrays that are Transient Variables or Masked Arrays to Numeric arrays. This is easily accomplished using the filled method. For example:

```
>>> import cdms, MV
>>> f = cdms.open('../data_directory/clt.nc')
>>> v = f('clt')
# If we want the Numeric array to have 1.e+20
# where the value is missing.
>>> v_array = MV.filled(v, 1.e+20)
```

Figure 2 summarizes the functions we have just seen.

FIGURE 2. Array functions.



3.1.3 Handling Time

The **cdtime** module implements the **cdms** time types, methods, and calendars. These are made available with the command

```
>>> import cdtime
```

Two time types are available: *relative time* and *component tim*e. Relative time is time relative to a fixed base time. It consists of:

- a ***units*** string, of the form **"units since basetime**", and
- a floating-point ***value***

For example, the time "28.0 days since 1996-1-1" has value=**28.0**, and units**="days since 1996-1-1"** Component time consists of the integer fields **year, month, day, hour, minute**, and the floating-point field **second**. A sample component time is **1996-2-28 12:10:30.0**

The **cdtime** module contains functions for converting between these forms, based on the common calendars used in climate simulation.

Basic arithmetic and comparison operators are also available. Some examples are shown below.

```
# Example: Creating component and relative times.
>>> import cdtime
>>> c = cdtime.comptime(1996,2,28)
>>> r = cdtime.reltime(28,"days since 1996-1-1")
# Example: Adding and subtracting times.
>>> print r.add(1,Days)
29.00 days since 1996-1-1
>>> print c.add(36,Hour)
1996-2-29 12:0:0.0
>>> print r.sub(10,Days)
18.00 days since 1996-1-1
>>> print c.sub(30,Days)
1996-1-29 0:0:0.0
# Example: Comparing times.
>>> r = cdtime.reltime(28,"days since 1996-1-1")
>>> c = cdtime.comptime(1996,2,28)
>>> print r.cmp(c)
-1
>>> print c.cmp(r)
1
>>> print r.cmp(r)
0
# Example: Extracting year/month/day information
>>> print c.year
1996
>>> print c.month
2
# Example: Converting between time types.
>>> r.tocomp()
1996-1-29 0:0:0.0
>>> print c.torel("days since 1996-1-1")
```

```
58.00 days since 1996-1-1
>>> print r.torel("days since 1995")
393.00 days since 1995
>>> print r.torel("days since 1995").value
393.0
```

More details of the **cdtime** module can be found in *Climate Data Management System* (**cdms.pdf**).

3.1.4 Axes and Domains

The spatial and temporal information associated with a variable is represented by the variable *domai*n, an ordered tuple of axes and/or grids. In the above example, the domain of the variable **u** is the tuple (time, latitude, longitude). This indicates the order of the dimensions, with the slowest-varying dimension listed first (time). Each element of the tuple is an *axi*s. An axis is like a 1-D variable, in that it can be sliced, and has attributes. A number of functions are available to access axis information. For example, to see the list of time values associated with **u**:

```
>>> t = u.getTime()
>>> print t[:]
[ 0., 366., 731.,]
# Similar methods of extracting the latitude, longitude
# and level axes or the lat-lon Grid are available
>>> myGrid = u.getGrid()
>>> lat_axis = u.getLatitude()
# Individual axes can also be accessed by their index
>>> first_axis = u.getAxis(0)
```

Similarly, creating axes and domains are easily accomplished using the constructor functions. Some of the basic constructors and methods are illustrated in the tutorials. More details of dealing with axes and domains can be found in *Climate Data Management System* (**cdms.pdf**).

3.1.5 Data Selection.

As seen previously, the **cdms** module is used to open files and extract data. The **cdms** module also allows for easy selection of subsets of the data stored in files or in memory. The use of keywords to describe specific axes, “Selectors” to describe specific portions of interest, and control over the precision of extracted areas are some of the features that make this a powerful package. A few simple examples are shown here.

```
>>> import cdms
>>> f = cdms.open(‘file_name’)
# To extract data for specified times
>>> ta_1996_only =f('ta',time=('1996-1-1','1996-12-1'))
# To extract data for specified latitude and
# longitude areas
>>> x =f('t', latitude=(-5.,5.), longitude=(210., 270.))
# Using cdutil to specify regions precisely.
>>> import cdutil
>>> NINO3 = cdutil.region.domain( \
              latitude=(-5.,5.), longitude=(210., 270.))
>>> nino3_area_exact = f('t', NINO3)
# In the above case, the precise region is returned with
# the weights and grid cell bounds set to match the
# request.
```

Some commonly used domains have been pre-defined for convenience. They are:

NH | NorthernHemisphere

SH | SouthernHemisphere

Tropics : latitude extends from -23.4 to 23.4

NPZ | AZ | ArcticZone : latitude extends from 66.6N to 90.0N

SPZ | AAZ | AntarcticZone : latitude extends from 90.0S to 66.6S

Example:

```
>>> from cdutil import region
>>> t_northern_hemisphere_only = f('t', region.NH)
```

The tutorial examples illustrate these features in more detail. For details of the available commands and their usage refer to *Climate Data Management System* (**cdms.pdf**).

3.1.6 Regridding Data

CDMS has functions to interpolate gridded data:

- from one horizontal (lat/lon) grid to another
- from one set of pressure levels to another
- from one vertical (lat/level) cross-section to another vertical cross-section.

The simplest method to regrid a variable from one horizontal, lat/lon grid to another is to use the **regrid** function defined for variables. This function takes the target grid as an argument, and returns the variable regridded to the target grid:

```
>>> import cdms
>>> f = cdms.open('../data_directory/ccc/perturb.xml')
# Read the data and check the shape of the data
>>> rlsf = f('rls')
>>> rlsf.shape
(4, 48, 96)
# Now choose a file that contains data in the
# desired output (target) grid.
>>> g = cdms.open('../data_directory/mri/perturb.xml')
# Get the file variable. The data is not actually read
# in when we use square bracket pairs like so:
>>> rlsg = g['rls']
# Get the target grid
>>> outgrid = rlsg.getGrid()
# Apply the regrid method to get the desired result.
>>> rlsnew = rlsf.regrid(outgrid)
>>> rlsnew.shape
```

```
(4, 46, 72)
>>> outgrid.shape
(46, 72)
```

A somewhat more efficient method is to create a regridder function. This has the advantage that the mapping is created only once and can be used for multiple arrays.The steps in this process are:

- Given an input grid and output grid, generate a regridder function.
- Call the regridder function on an array, resulting in an array defined on the output grid. The regridder function can be called with any array or variable defined on the input grid.

The following example illustrates this process.

```
>>> import cdms
>>> from regrid import Regridder
>>> f = cdms.open('../data_directory/ccc/perturb.xml')
>>> rlsf = f['rls']
>>> ingrid = rlsf.getGrid()
>>> g = cdms.open('../data_directory/mri/perturb.xml')
>>> outgrid = g['rls'].getGrid()
>>> regridfunc = Regridder(ingrid, outgrid)
>>> rlsnew = regridfunc(rlsf)
>>> f.close()
>>> g.close()
```

For more details on regridding functions and methods please refer to *Climate Data Management System* (**cdms.pdf**).

3.1.7 Working with masks

Masks were introduced earlier in "Reading, creating, and altering variables" on page 33. They are a convenient way to deal with data that either has missing values or where one would have to deal with masking out regions. The masks can be handled seperately from the data to keep the computational expense down and one can also use the logical and/or operators to perform complex tasks easily. A small example is shown below. More details of how to use masks are in the Numeric manual and examples of masks and masking operations in action can be found in the geting started tutorials listed in Chapter 2.

```
# Let us open a data file that contains surface type
# (land fraction) data and extract data
>>> import cdms, MV
>>> f_surface = cdms.open('sftlf_ta.nc')
>>> surf = f_surface('sftlf')
# Designate land where “surf” has values
# not equal to 100
>>> land_only = MV.masked_not_equal(surf, 100.)
>>> land_mask = MV.getmask(land_only)
# Now extract a variable from another file
>>> f = cdms.open('ta_1994-1998.nc')
>>> ta = f('ta')
# Apply this mask to retain only land values.
>>> ta_land = cdms.createVariable(ta, mask=land_mask,
      copy=0, id='ta_land')
```

3.1.8 Databases

A Database is a collection of datasets and other CDMS objects. It consists of a hierarchical collection of objects, with the database being at the root, or top of the hierarchy. A database is used to:

- search for metadata
- access data
- provide authentication and access control for data and metadata

Details of creating, altering, and searching through databases are beyond the scope of this beginners document. The interested reader should refer to *Climate Data Management System* (**cdms.pdf**).

3.2 Averaging and Statistics

3.2.1 Area averaging

Area averaging is one of the most common data reduction procedures used in climate data analysis. The **cdutil** package has a powerful area averaging function. The averager() function provides a convenient way of averaging your data giving you control over the order of operations (i.e which dimensions are averaged over first) and also the weighting for the different axes. You can pass your own array of weights for each dimension or use the default (grid) weights or specify equal weighting.

Examples:

```
>>> import cdms, cdutil
>>> f = cdms.open('data_file_name')
>>> result = cdutil.averager(f('var_name'), axis='1')
# extracts the variable 'var_name' from f
# and averages over the dimension whose position is 1.
# Since no other options are specified,
# defaults kick in i.e weights='generate' (same as
# weights='weighted') and returned=0
# Some ways of using the averager are shown below.
#
# A quick zonal mean calculation would be:
>>> V_zonal_ave = cduitl.averager(V, axis='x')
# In the above case, default weights option of
# 'generate' (or 'weighted') is implemented
#
# If you want to average first over the x (longitude)
# dimension with area weighting and then over
# y (latitude) with equal weighting, then you would:
>>> Vavg = cduitl.averager(V, axis='xy', \
      weight=['generate','equal'])
# Similarly for equally weighted time averaging:
>>> cduitl.averager(V, axis='t', weight='equal')
#
>>> cduitl.averager(V, axis='x', weight=mywts)
```

```
# where mywts is an array of shape (len(xaxis))
# or shape(V)
#
>>> cduitl.averager(V, axis='(lon)y', weight=[myxwts,
      myywts])
# where myxwts is of shape len(xaxis) and
# myywts is of shape len(yaxis)
#
>>> cduitl.averager(V, axis='xy', weight=V_wts)
# where V_wts is a Masked Variable of shape V
# or
>>> cduitl.averager(V, axis='x', weight='equal',
      action='sum')
# will return the equally weighted sum
# over the x dimension or
>>> ywt = cduitl.area_weights(y)
>>> fractional_area= cduitl.averager(ywt, axis='xy',\
                     weight=['equal','equal'],\
                     action='sum')
# is a good way to compute the area fraction that the
# data y that is non-missing
```

Note: When averaging data with missing values, extra care needs to be taken. It is recommended that you use the default weight='generate' option.   This uses cdutil.area_weights(V) to get the correct weights to pass to the averager.

```
>>> cduitl.averager(V, axis='xy', weight='generate')
# The above is equivalent to:
>>> V_wts = cdutil.area_weights(V)
>>> result = cduitl.averager(V, axis='xy', weight=V_wts)
#
>>> result = cduitl.averager(V, axis='xy',
      weight=cdutil.area_weights(V))
```

The following example will help you see the averager() function in context

```
>>> import cdms, cdutil
>>> f = cdms.open('file_name')
# Using cdutil.domain to specify the NINO3 region
>>> NINO3 = cdutil.domain(latitude=(-5.,5.),\
      longitude=(210.,270.)))
# Extract the variable over the specified domain
>>> nino3_area_exact = f('t', NINO3)
# Average first over the longitude axis
# (denoted by 'x') and then the latitude axis
# (denoted by 'y')
>>> nino3_avg = cdutil.averager(nino3_area_exact,
      axis='xy')
```

Axis options can also be specified by name such as axis = '(depth)' or by index such as axis = '20' (note the numbers are enclosed in quotes). By default, the appropriate area weights are generated from the grid information and the result of the averaging is the area weighted value. More control over the weights used is available. It is possible to specify the weights used to average over the longitude and latitude axes seperately.

```
>>> nino3avg2 = cdutil.averager(nino3_area_exact,
      axis='yx',weights=['generate','equal'])
```

In the above example, we averaged over the latitude axis first (using generated weights) and then over the longitude axis (using equal weights). The weights can be "equal" or "generate"(generates the weights for the grid information contained in the variable) or any array of numbers the user wishes to apply.

3.2.2 Generating weights

For most averaging applications, the weights used are critical especially when there are missing data. The **cdutil** package provides a way of generating the weights using grid information that is tied to the variable. The averager function uses this to generate the weights when the default averaging weights option kicks in. This function is easily called for some variable 'x' in memory:

```
>>> gen_weights = cdutil.area_weights(x)
```

The resultant gen_weights is in the same shape as the variable x and has the appropriate area weights set to missing values where data was missing in x.

3.2.3 Time averaging

Averaging over time is a special problem in climate data analysis. The **cdutil** package pays special attention to this issue to make the extraction of time averages and climatologies simple. Apart from functions that enable easy computation of annual, seasonal and monthly averages and climatologies, one can also define seasons other than those already available and specify criteria for data availability and temporal distribution to suit individual needs.

Note: It is essential that the data have an appropriate axis designated as the “time” axis. In addition to this, the results depend on the time axis having correctly set “bounds”. If “bounds” are not stored with the data in files, default “bounds” are generated by the data extraction steps in **cdms**. However, they are not always correct. The user must take care to verify that the bounds are set correctly.

The predefined time averaging periods are:

- JAN, FEB, MAR, ..., DEC (months)
- DJF, MAM, JJA, SON (seasons)
- YEAR (annual means)
- ANNUALCYCLE (monthly means for each month of the year)
- SEASONNALCYCLE (means for the 4 predefined seasons)

Some simple examples of time averaging operations are shown here.

```
>>> import cdutil
# To compute the DJF (december-January-February)
# climatology of a variable x
```

```
>>> djfclim = cdutil.DJF.climatology(x)
# The individual DJF seasons are extracted using
>>> djfs = cdutil.DJF(x)
# To extract DJF seasonal anomalies (from climatology)
>>> djf_anom = cdutil.DJF.departures(x)
# The monthly anomalies for x are computed by:
>>> x_anom = cdutil.ANNUALCYCLE.departures(x)
```

3.2.3.1 Creating Custom Seasons

You can even create your own “custom seasons” beyond the pre-defined seasons listed above. For example:

```
>>> JJAS = cdutil.times.Seasons(‘JJAS’)
```

3.2.3.2 Specifying time periods for climatologies

So far we have seen the way to compute the means, climatologies, and anomalies for the entire length of the time-series. The typical application may require specified time intervals over which climatologies are computed and used in calculating departures. For example, to compute the DJF climatology for the time period 1979-1988 we would do the following:

```
>>> import cdtime
>>> start_time = cdtime.comptime(1979)
>>> print 'start_time = ', start_time
>>> end_time = cdtime.comptime(1989)
>>> print 'end_time = ', end_time
```

Note that we created the time point 'end_time' at the begining of 89 so we can select all the time between 'start_time' and 'end_time' but not including 'end_time' by specifying the option 'co' - shorthand for 'c'losed at start_time and 'o'pen at end_time. For more details on different options available, refer to *Climate Data Management System* (**cdms.pdf**).

```
>>> djfclim = cdutil.DJF.climatology(x(time= \
      (start_time, end_time, 'co')))
```

Now that we have our climatology over the desired period we can to compute anomalies over the full period relative to that climatology.

```
>>> djfdep2 = cdutil.DJF.departures(s, ref=djfclim)
```

3.2.3.3 Specifying Data Coverage Criteria

The real power of these functions is in the ability to specify minimum data coverage and to also be able to specify the distribution (both in the temporal sense) which are *required* for the averages to be computed. The default behaviour of the functions that compute seasonal averages, climatologies etc. is to require that a minimum of 50% of the data be present. Now let's say you like to extract DJF but without restricting it to 50% of the data being present. You would do:

```
>>> djfs = cdutil.DJF(avg, criteriaarg=[0., None])
```

The above statement comutes the DJF average with "criteriaarg" (passed as a list) which has 2 arguments.

- The first argument represents the *minimum* fraction of time that is required to compute the seasonal mean. So you can pass a fractional value between 0.0 and 1.0 (including both extremes) or even a representation such as 3.0/4.0 (in case you need at least 3 out of 4 months of data in the case of the average JJAS we defined previously).
- The second argument in the criteriaarg is "None". This implies no "centroid function" is used. In other cases this argument represents the *maximum* value of the "centroid function".A value between 0 and 1 represents the spread of values across the mean time. The centroid value of 0.0 represents a full even distribution of data across the time interval. For example, if you are considering the DJF average, then if data is available for Dec, Jan and Feb months then the centroid is 0.0. On the other hand, the following criteria will "mask"(i.e ignore) a DJF season if there is only a december month with data (and therefore has a centroid value of 1.0). Therefore any seasons resulting in centroid values above 0.5 will result in missing values!

```
>>> djfs = cdutil.DJF(avg, criteriaarg = [0., .5])
```

In the case of computing an annual mean, having data only in Jan and Dec months leads to a centroid value of 0 for the regular centroid, and the resulting annual mean for the year is biased toward the winter. In this situation, you should use a cyclical centroid where the circular nature of the year is recognised and the centroid is calculated accordingly. Here are some examples of typical usage:

1) Default behaviour i.e criteriaarg=[0.5, None]

```
>>> annavg_1 = cdutil.YEAR(s_glavg)
```

2) Criteria to say compute annual average for any number of months.

```
>>> annavg_2 = cdutil.YEAR(s_glavg, criteriaarg =
      [0.,None])
```

3) Criteria for computing annual average based on the minimum number of months (8 out of 12).

```
>>> annavg_3 = cdutil.YEAR(s_glavg, \
          criteriaarg = [8./12., None])
```

4) Same criteria as in 3, but we account for the fact that a year is cyclical i.e Dec and Jan are adjacent months. So the centroid is computed over a circle where Dec and Jan are contiguous.

```
>>> annavg_4 = cdutil.YEAR(s_glavg, \
       criteriaarg = [8./12., 0.1, 'cyclical'])
```

So far we have the annual means calculated using various criteria. Now if we wish to compute the climatological annual mean, we can average the individual annual means. However, we can apply more criteria to the calculation of that annual mean climatology. Here we simply require 60% of the years to be present, and a criteria on the temporal distribution (i.e the centroid = 0.7) to make sure all of the annual means are not clustered at the end of the record.

```
>>> annavg_clim = cdutil.YEAR.average(annavg_4,\
      criteriaarg =[.6,.7])
```

The tutorial file *times_tutorial.py* has detailed examples of time averaging in action. Further documentation is available in the CDAT Utilities Reference Manual cdat_utilities.pdf.

3.2.4 Useful statistical functions

Commonly used statistical functions such as corrrelation, covariance, autocorrelation, autocovariance, laggedcorrelation, laggedcovariance, rms, variance, standard deviation, mean absolute difference, geometric mean, and linearregression have been implemented to allow easy computation of statistics. The statistics functions are implemented as part of the **genutil** package. These functions are implemented so as to not require the full variable information in MV. That is, these functions accept Numeric arrays. However, they also accept MV's so that the user can specify axes over which statistics are computed (to allow for spatial or temporal statistics etc.). The tutorial file *statistics_tutorial.py* shows some of the statistics functions in action.

Example 1

Let us try an example where we want to look at a variable 'tas' from the NCEP reanalysis and compute some spatial statistics between data slices for time periods from 1960-1970 and 1980-1990.

```
>>> import cdms
>>> from genutil import statistics
>>> f = cdms.open('tas.rnl_ncep.nc')
>>> ncep1 = f('tas',time=(‘1960-1-1’, ‘1970-1-1’, 'co'))
>>> ncep2 = f('tas',time=(‘1980-1-1’, ‘1990-1-1’, 'co'))
# We have the two time periods extracted.
# Now let us compute the correlation.
>>> cor = statistics.correlation(ncep1, ncep2,\
            axis=’xy’)
# We could compute the spatial correlation weighted by
# area. To accomplish this we can use the ‘generate’
# option for weights.
>>> wcor = statistics.correlation(ncep1, ncep2,\
```

```
weights='generate', axis='xy')
```

Example 2

To compute the mean absolute difference between ncep1 and ncep2 defined in Example 1.

```
>>> absd = statistics.meanabsdiff(ncep1, \
      ncep2,axis='xy')
```

Example 3

To compute the "temporal" rms difference between the two time periods

```
>>> rms = statistics.rms(ncep1, ncep2, axis='t')
```

Example 4

In this example, we examine the default behaviour of the linearregression function.

```
>>> Values = statistics.linearregression(y)
```

The returned "Values" is actually a tuple consisting of the slope and itercept. They can also be accessed as follows:

```
>>> slope, intercept = statistics.linearregression(y)
```

If error estimates are also required, then:

```
>>> Values, Errors = linearregression(y, error=1)
```

where "Values" and "Errors" are tuples containing answer for slope AND intercept. You can break them as follows. slope, intercept = Value and slope_error, intercept_error = Errors. i.e.

```
>>> (slope, intercept), (slope_error, intercept_error) =
      \ linearregression(y, error=1)
```

WARNING: The following will not work.

```
>>> slope, intercept, slo_error, int_error =
      linearregression(y, error=1)
```

To get the standard error non adjusted result for slope only, do the following:

```
>>> slope, slope_error = linearregression(y, error=1,
      nointercept=1)
```

In the line below all the returned values are tuples.

```
>>> Values,Errors,Pt1,Pt2,Pf1,Pf2 = \
      linearregression(y, error=2,probability=1)
```

That means in the above statement is returning tuples ordered so: (slope, intercept), (slo_error, int_error), (pt1_slo, pt1_int), (pt2_slo, pt2_int), (pf1_slo, pf1_int), (pf2_slo, pf2_int).

If we want results returned for the intercept only, do the following:

```
>>> intercept,intercept_error,pt1,pt2,pf1,pf2=\
 linearregression(y,error=2,probability=1,noslope=1)
```

3.3 Data Visualization

Data visualization is a very important aspect of analysing climate data. Visualization as a part of analysis and creating presentation quality graphics are accomplished in CDAT by using the point and click GUI in VCDAT, from the CDAT command line, or in a program. The primary tool inside CDAT to visualize the data is the Visualization and Control System (VCS). Additionally, because of the ease of controlling other applications using Python as a link language, an interface to Grace (an open source application) is also available to users.

3.3.1 Visualization and Control System (VCS)

VCS is expressly designed to meet the needs of climate scientists. Because of the breadth of its capabilities, VCS can be a useful tool for other scientific applications as well. VCS allows wide-ranging changes to be made to the data display, provides for presentation hardcopy output, and includes a means for recovery of a previous display.

In the VCS model, the data display is defined by a trio of named object sets, designated the "primary objects" (or "primary elements"). These include:

- the data, which define what is to be displayed and is ingested via other CDAT software components;
- the graphics method, which specifies the display technique; and
- the picture template, which determines the appearance of each segment of the display.

Tables for manipulating these primary objects are stored in VCS for later recall and possible use. In addition, detailed specification of the primary objects' attributes is provided by eight "secondary objects" (or "secondary elements"):

1. *colormap:* specification of combinations of 256 available colors
2. *fill area:* style, style index, and color index
3. *format:* specifications for converting numbers to display strings
4. *line:* line type, width and color index
5. *list:* a sequence of pairs of numerical and character values
6. *marker:* marker type, size, and color index
7. *text:* text font type, character spacing, expansion and color index
8. *text orientation:* character height, angle, path, and horizontal/vertical alignment

By combining primary and secondary objects in various ways (either at the command line or in a program), the VCS user can comprehensively diagnose and inter-compare climate model simulations. VCS provides capabilities to:

- Create and modify existing template attributes and graphics methods
- Save the state-of-the-system as a script to be run interactively or in a program
- Save a display as a Computer Graphics Metafile (CGM), GIF, Postscript, Sun Raster, or Encapsulated Postscript file
- Create and modify colormaps
- zoom into a specified portion of a display
- Change the orientation (portrait vs. landscape) or size (partial vs. full-screen) of a display
- Animate a single data variable or more than one data variable simultaneously
- Display different map projections

3.3.2 Displaying data

VCS can handle the CDMS data objects (such as Transient Variables seen earlier inaddition to Numeric arrays, MA arrays, python lists, and tuples). Plotting data is simply accomplished by importing the **vcs** module and issuing the plot command. For example

```
>>> import cdms, vcs
>>> f =cdms.open('example.nc')
>>> my_data = f('clt')
>>> v = vcs.init()
>>> v.plot(my_data)
```

The plot will display the data “my_data” using default settings. However, the user can control every aspect of the plot’s appearance individually. The first of those is the “Graphics Method” described in the next section.

3.3.2.1 Graphics Methods

A graphics method simply defines how data is to be displayed on the screen. Currently, there are eleven different graphics methods with more on the way. Each graphics method has its own unique set of attributes (or members) and functions. They also have a set of core attributes that are common in all graphics methods. The descriptions of the current set of graphics methods are as follows:

- boxfill - The boxfill graphics method draws color grid cells to represent the data on the VCS Canvas.
- isofill - The isofill graphics method fills the area between selected isolevels (levels of constant value) of a two-dimensional array with a user-specified color.
- isoline - The isoline graphics method draws lines of constant value at specified levels in order to graphically represent a two-dimensional array. It also labels the values of these isolines on the VCS Canvas.
- outfill - The outfill graphics method fills a set of integer values in any data array. Its primary purpose is to display continents by filling their area as defined by a surface type array that indicates land, ocean, and sea-ice points.
- outline - The Outline graphics method outlines a set of integer values in any data array. Its primary purpose is to display continental outlines as defined by a surface type array that indicates land, ocean, and sea-ice points.
- scatter - The scatter graphics method displays a scatter plot of two 4-dimensional data arrays, e.g. A(x,y,z,t) and B(x,y,z,t).
- vector - The Vector graphics method displays a vector plot of a 2D vector field. Vectors are located at the coordinate locations and point in the direction of the data vector field. Vector magnitudes are the product of data vector field lengths and a scaling factor.
- xvsy - The XvsY graphics method displays a line plot from two 1D data arrays, that is X(t) and Y(t), where ‘t’ represents the 1D coordinate values.
- xyvsy - The Xyvsy graphics method displays a line plot from a 1D data array, i.e. a plot of X(y) where ‘y’ represents the 1D coordinate values.

- yxvsx - The Yxvsx graphics method displays a line plot from a 1D data array, i.e. a plot of Y(x) where ‘x’ represents the 1D coordinate values.

Say for example you would like to plot the data object “my_object” using the “isofill” graphics method (instead of the default “boxfill” method), you would type:

```
>>> v.isofill(my_data)
```

The user can control any aspect of the isofill method to get the precise appearance on the plot. To alter the isofill methods for use in your plot it will be necessary to “create” an isofill object. The create functions allow the user to create VCS objects which can be modified directly to produce the desired results. Since the VCS “default” objects allow for modifications, it is best to either create a new VCS object or get an existing one. When a VCS object is created, it is stored in an internal table for later use and/or recall.

```
>>> my_new_isofill = v.createisofill(‘newisofillname’)
```

To show the list of existing isofill graphics methods type:

```
>>> v.show(‘isofill’)
```

If there is an existing isofill method you have created and would like to use it (or alter it), then type:

```
>>> my_old_isofill =
      v.getisofill(‘existing_isofillobjectname’)
```

or use this “existing_isofillname” as a base to create the new one:

```
>>> new_isofill2 = v.createisofill(‘newisofillname’,\
      ’existing_isofillobjectname’)
```

The list of attributes can be viewed with the following command:

```
>>> my_old_isofill.list()
```

You can explicitly set each of the attributes. For example, to change the levels to be used in plotting:

```
>>> my_old_isofill.levels = [2., 3., 4., 6]
```

The graphics method can also be used as an optional argument to the plot command so that VCS now allows you to issue the command:

```
>>> v.plot(my_data, my_old_isofill)
# You can also pass the name of the graphics
# method object or the object itself as shown below
>>> v.plot(my_data, “isofill”, “isofill_name”)
```

To enable the user to check whether an object is of a certain graphics method (or any VCS object), there are a whole set of query functions made available. Therefore you can check if my_old_isofill is an isofill object and get a yes/no (1|0) answer by saying:

```
>>> my_old_isofill.isisofill()
```

For help with queries type (at the shell prompt):

```
% pydoc vcs.queries
```

For more information you can type:

```
>>> vcs.help(‘plot’)
```

or

```
>>> print v.plot.__doc__
```

Extensive documentation can be found in *Visualization and Control System: Command Line and Application Programming Interface* (**vcs.pdf**).

3.3.2.2 Graphics Primitives

Graphics primitives are useful in enhancing plots and in creating additional ways of displaying data beyond the existing graphics methods. These primitive objects are created and altered using the same syntax as in the case of graphics methods. The following primitive objects are available:

- fillareaobject - The fillarea objects allows the user to edit fillarea attributes, including fillarea interior style, style index, and color index. The fill area attributes are used to display regions defined by closed polygons, which can be filled with a uniform color, a pattern, or a hatch style. Attributes specify the style, color, position, and dimensions of the fill area.
- lineobject – The line object allows the editing of line type, width, and color index. The line attributes specify the type, width, and color of the line to be drawn for a graphical display.
- markerobject – The marker object allows the editing of the marker type, width, and color index. The marker attribute specifies graphical symbols, symbol sizes, and colors used in appropriate graphics methods.
- textobject - Graphical displays often contain textual inscriptions, which provide further information. The text-table object attributes allow the generation of character strings on the VCS Canvas by defining the character font, precision, expansion, spacing, and color. The text-orientation object attributes allow the appearance of text character strings to be changed by defining the character height, up-angle, path, and horizon tal and vertical alignment. The text-combined object is a combination of both text-table and text-orientation objects.

3.3.2.3 Templates

A picture template determines the location of each picture segment, the space to be allocated to it, and related properties relevant to its display. The description of the picture template is as follows:

- template - Picture Template attributes describe where and how segments of a picture will be displayed. The segments are graphical representations of: textual identification of the data formatted values of single-valued dimensions and mean, maximum, and minimum data values axes, tick marks, labels, boxes, lines, and a legend that is graphics-method specific the data. Picture templates describe where to display all segments including the data.

Templates also can be created and altered just like the graphics methods seen in the previous section. The syntax is retained the same for all objects in VCS so as to avoid confusion. The general philosophy behind templates is - "You should be able to specify the behaviour of every picture segment - text, data, line, etc. precisely according to your needs."

By setting the "priority" attribute of each picture segment you can control the order in which segments are drawn and whether they are drawn at all. The higher the priority number (integer value), the later it is drawn during plot creation ensuring that it is on top. Setting priority = 0 means you do not want it drawn!

The following segments of a template can be controlled and the values that can be set are.

- **data :** The location where the data is plotted (x1, x2, y1, y2) and its priority can be specified.
- **title :** The location(x, y), priority and the text objects (font type, size, angle, justification, etc. etc.) (More on text objects later). The title plotted is the title of your data read in. If you are plotting the variable “tdata”, tdata.title is shown in the location specified for title. You can alter the title by doing:

```
>>> tdata.title = 'My new title!!’
```

- **units :** The data units. For example: "Degrees C" You can set the x,y location, priority and text object (more later on text objects).
- **dataname :** The name of the variable.
- **source :** The data source description.
- **function :** If computed data, the user can give the algebraic equation.
- **file :** The location of the file.

- **crdate :** The date of creation of plot. You can control x,y location, priority, and text object.
- **crtime :** The time of creation of plot. You can control x,y location, priority, and text object.
- **mean, max, min :** The values are computed automatically from the data you are plotting and you can set the x,y location, text object, priority and display format.
- **legend :** You can set the legend bar location and dimensions (x1, x2, y1, y2), the line type object, text object and of course priority.
- **comment1, comment2, comment3, comment4 :** Four text comments can be drawn. For each of these you can set priority, location(x, y), and text object.You would have to set the comment on the data. Say you are plotting "tdata" using the x.plot(tdata) command, then you need to have done:

```
>>> tdata.comment1 = 'Your specified comment1"
```

- **line1, line2, line3, line4 :** Four lines can be drawn. For each of these lines you can set priority, start location (x1, y1), end location (x2, y2) and line object.
- **box1, box2, box3, box4 :** Four boxes can be drawn. Same as line except the x1, x2, y1, y2 settings refer to corners of the box.
- **xname, yname, zname, tname :** These are the possible axes of x,y,z and t. Note in the case you are trying the plot is a latitude x longitude plot. If it was a timexlongitude plot then you would be setting the xname and tname values. You can set the x,y location for the name, the priority and text object.
- **xunits, yunits, zunits, tunits :** These are the respective axis units for whichyou can set the x,y location, text object, and priority .
- **xvalue, yvalue, zvalue, tvalue :**
- **xlabel1, xlabel2 :** The x axis labels (bottom and top of your plot) can be independently set. You can specify the y location (x is determined by the data), priority and the text object.
- **ylabel1, ylabel2 :** Same as above except for the left and right side of your plot. Here you can only specify the x location of the label.
- **xtic1, xtic2 :** The major tic marks on the bottom and top. You can control the priority, y1 and y2 (in effect specifying the length!), and line object

(Things like line style, thickness, arrow, etc. etc. More on the line object later).

- **xmintic1, xmintic2:** The minitic specifications. Exactly the same as above otherwise.
- **ytic1, ytic2 :** The major tic marks on the left and right. You can control the priority, x1 and x2 (in effect specifying the length!), and line object.
- **ymintic1, ymintic2:** The minitic specifications. Exactly the same as above otherwise.

3.3.2.4 Animation

VCS allows the user to animate the contents of the VCS Canvas. This function pops up the animation GUI which lets the user control all aspects of the animation.

```
>>> v.plot(array,'default','isofill','quick')
>>> v.animate.gui()
```

Alternately, animation can be controlled from the command line using the following methods:

```
>>> v.animate.create()
>>> v.animate.run()
>>> v.animate.zoom(3)
>>> v.animate.horizontal(50)
>>> v.animate.vertical(-50)
>>> v.animate.pause(4)
>>> v.animate.frame(2)
>>> v.animate.stop()
```

3.3.2.5 Output

Before attempting to print your plots, make sure that gplot is built and installed on your system. The VCS graphics can be output to files of various formats or directly to printers with postscript capability. The available graphics file formats that one can print to are postscript, encapsulated postscript (EPS), GIF, CGM, and raster. To print directly to a printer and optionally specifying a portrait orientation:

```
>>> v.printer('printer_name', 'p')
```

To create gif, postscript, cgm, raster, and encapsulated postscript files, the commands are :

```
>>> v.gif('example.gif','r','p')
>>> v.postscript('example.ps', 'r', 'p')
>>> v.cgm('example.cgm', 'p')
>>> v.raster('example.ras', 'p')
>>> v.eps('example.eps', 'r', 'p')
```

3.3.2.6 Using VCS Scripts

Script commands define the actions that are necessary to preserve an interactive session as a script and to mimic that session in a non-interactive replay of the script. Many attributes are needed to create a graphical representation of a variable, e.g. attributes to identify the variable and to label the plotting axes. By use of VCS and Python scripts, most of these attributes can be manipulated to create the desired visual effect, and the resulting attributes can be saved for later use. VCS and Python scripts also allow the user to save a sequence of interactive operations for replay, and to recover from a system failure. To re-save the initial.attributes file, use the function saveinitialfile(). To save VCS objects as Python scripts or VCS scripts, use the function scriptobject(). To save the state of the system, use the function scriptstate(). To run a VCS script file, use the function scriptrun()

3.3.3 Interface to Grace (genutil.xmgrace)

Nothing emphases the fact that CDAT is a collection of tools that can be extended by the user better than the **xmgrace** module. This module provides an interface to the popular Grace plotting utility (which you must have installed separately. Downloads and information are available from http://plasma-gate.weizmann.ac.il/Grace). The xmgrace tutorial which is introduced in "xmgrace_tutorial.py" on page 27 teaches you how to use it.

3.4 Other useful packages

The following packages are contributions from users. The module descriptions are shown in the CDAT Utilities Reference Guide cdat_utilities.pdf. They are provided "as-is" and may not be supported unless the package is considered useful by a large user community.

3.4.1 Interface to Spherepack

This package contains a Python interface to the subroutine library Spherepack. To see list of functions type

```
% pydoc -w sphere
```

3.4.2 Interface to Regridpack

This package contains a Python interface to the subroutine library regridpack. For further details type:

```
% pydoc -w adamsregrid
```

3.4.3 Empirical Orthogonal Functions

Available in the **eof** package. Calculates Empirical Orthogonal Functions of either one variable or two variables jointly. For more documentation type:

```
% pydoc -w eof.
```

3.4.4 Interface to the L-moments library

An interface to an L-moments library by J. R. M. Hosking. To see list of functions type:

```
% pydoc -w lmoments
```

3.4.5 Interface to the ngmath library

The ngmath library is a collection of interpolators and approximatorsfor one-dimensional, two-dimensional and three-dimensional data. The packages, which were obtained from NCAR, are:

- natgrid - a two-dimensional random data interpolation package based on Dave Watson's nngridr. NOT built by default in CDAT due to compile problems on some platforms. Works on linux.
- dsgrid - a three-dimensional random data interpolator based on a simple inverse distance weighting algorithm.
- fitgrid - an interpolation package for one-dimensional and two-dimensional gridded data based on Alan Cline's Fitpack. Fitpack uses splines under tension to interpolate in one and two dimensions. NOT IN CDAT.
- csagrid - an approximation package for one-dimensional, two-dimensional and three-dimensional random data based on David Fulker's Splpack. csagrid uses cubic splines to calculate its approximation function.

3.4.6 Using existing Fortran code

3.4.6.1 Pyfort

Pyfort is a tool for connecting Fortran (Fortran90) routines to Python (www.python.org). Pyfort translates an input file that describes the Fortran functions and subroutines you wish to access from Python into a C language source file defining a Python module. Fortran was changed significantly by the introduction of the Fortran

90 standard. We will use the phrase "modern Fortran" to indicate versions of Fortran from Fortran 90 onwards. Pyfort's input uses a syntax that is a subset of the modern Fortran syntax for declaring routines and their arguments. The current release does not yet support modern Fortran's "explicit-interface" routines. However, the tool was designed with this in mind for a future release. Pyfort can in most cases also build and install the extension you create.

The Pyfort project page at SourceForge contains documentation and releases. It is: http://sourceforge.net/projects/pyfortran

3.4.6.2 F2PY (previously known as fpig)

Writing Python C/API wrappers for Fortran routines can be a very tedious task, especially if a Fortran routine takes more than 20 arguments but only few of them are relevant for the problems that they solve. Pearu Petersen has developed a tool that generates the C/API modules containing wrapper functions of Fortran routines. This tool is called F2PY - Fortran to Python Interface Generator. It is completely written in Python language and can be called from the command line as f2py. F2PY is released under the terms of GNU LGPL. The F2PY package and documentation can be downloaded from http://cens.ioc.ee/projects/f2py2e/

3.4.7 Migrating from GrADS (grads)

The grads module supplies an interface to CDMS that will be familiar to users of GrADS. See the CDAT website for documentation.

3.4.8 ort

Read data from an Oort file.

3.4.9 trends

Computes variance estimate taking auto-correlation into account.

CHAPTER 4 Contributions to CDAT

CDAT is a collaboration. You can be part of the collaboration. You don't need permission or PCMDI's approval. You don't need PCMDI's programmers to add your algorithms. Here are descriptions of how to contribute packages the packages contributed to the "contrib" section of the CDAT source.

4.1 How to add your packages

One of CDAT's strengths is that it is an open system. You can add your own software written in C, Python, or Fortran. The easiest way to learn to do this is to copy our examples. Get the CDAT source distribution and look for subdirectory 'contrib' in the top-level directory. The README file in contrib explains what to do.

There are tools that may be useful to you.

- The SWIG utility (Simplified Wrapper and Interface Generator, **http://www.swig.org**) can wrap C and C++ routines.
- Pyfort (**http://pyfortran.sourceforge.net**) connects Fortran routines to Python.

Depending on your needs, you may wish to use a layer of Python along with the automatically created interface, in order to make a nicer interface or to use the Fortran or C simply as computational engines. An example of this is the EOF package described below: it uses a Fortran linear algebra routine to enhance performance, but the "science" is in Python.

If you follow the protocols in '**contrib**' then your package can be added to the PCMDI distribution as well. Just send it to us and be sure to include a README that explains:

- How to use the package
- Contact information about the author.

You may also be able to generate useful documentation by executing the routines happydoc or pydoc. happydoc works only on Python code; pydoc works on the installed modules. Both routines print help packages if executed with the argument, '--help', and both are already installed in your cdat 'bin' directory.

If you have the source distribution, use the README files in the sub-directories of the contrib directory for full documentation. Alternately, type

```
% pydoc -w <name_of_package>
```

to create a web page showing the package's interface.

Index

A

adding your packages 73
Animate 59
Animation 66
Area averaging 48
arrays 36
ASCII text files 33
asciidata 33
autocorrelation 55
autocovariance 55
axis 43

B

base time 41
binaryio 33
boxfill 60
bug-tracking facility 30

C

CDAT Home Page 2, 30
CDAT Website 30
cdms 33
cdtime 41, 42
cdutil 48, 50
centroid function 53
CGM 59
colormap 58, 59
component time 41
contrib 74
corrrelation 55
covariance 55
criteriaarg 53
custom seasons 52

D

data, conversion to Numeric 40
databases 47
docstrings 28

documentation 27
documentation, run-time 74
domain 34, 43

E
Empirical Orthogonal Functions 68
Encapsulated Postscript 59
eof (package) 68

F
F2PY 70
File I/O. 31
file variable 35
fill area 58
fillareaobject 63
format 58
Fortran formatted data 32
fpig 70

G
Generating weights 50
genutil 55
geometric mean 55
GIF 59
Grace 57, 67
GrADS 70
grads (module) 70
GrADS/GRIB 31
graphics method 58
Graphics Methods 60
Graphics Primitives 63

H
happydoc 29, 74
HDF 31

I
isofill 60
isoline 60

L
laggedcorrelation 55

laggedcovariance 55
Learning Python 17
line 58
linearregression 55
lineobject 63
list 58
L-moments 69

M

MA 36
mailing list 30
marker 58
markerobject 63
Masked Array 35
masked arrays 36
Masked Variable 35
masks 46
mean absolute difference 55
metadata 34
missing values 36
MV 36

N

netCDF 31
Numeric 36
Numeric array 35, 37
Numerical Python 27, 37

O

orientation 59
ort (package) 70
outfill 60
outline 60

P

picture template 58
Postscript 59
primary objects 58
projections 59
pydoc 28, 74
Pyfort 69
Python 5

Python scripts 67

R
Raster 59
regrid 45
regridder function 46
Regridding 45
Regridpack 68
regridpack (package) 68
relative time 41
rms 55

S
scatter 60
Scientific Python 32
Script 67
secondary objects 58
Selectors 44
self-describing formats 31
sphere (package) 68
Spherepack 68
standard deviation 55
statistical functions 55
statistics_tutorial.py 55
SWIG 73
System Requirements 3

T
Templates 63
text 58
text orientation 58
textobject 63
Time averaging 51
times_tutorial.py 55
transient variable 35
trends (package) 71

U
unformatted data 33

V
variable 34

variables 36
variance 55
VCDAT 5, 28
VCS 57, 58
VCS scripts 67
VCS, scripting 67
vector 60
Visualization 57
Visualization Control System 58

X

xmgrace (package) 67
xml 31
xvsy 60

Z

zoom 59