URL: http://cxc.harvard.edu/ciao3.4/slang−tips.html
Last modified: December 2006

AHELP for CIAO 3.4

tips

Context: slang

Jump to: Description Examples See Also

Synopsis

S−Lang tips and example functions

Description

This document presents a selection of short S−lang routines, along with a brief explanation. They are intended to provide "simple examples" which can be used for your own scripts; they therefore do not necessarily contain the safety checks which would be present in actual code. It is suggested that the S−Lang documentation at http://www.s−lang.org/ also be read. The source code for these routines can be obtained from the Scripts download page.

The documentation for the S−Lang run−time library is available from ahelp: use "ahelp slangrtl" or "ahelp −c slangrtl" to get a list of topics.

Using ChIPS and Sherpa as a calculator

The S−Lang interpreter can be used as a calculator at the Sherpa and ChIPS command−line prompt, since the result of a calculation is printed to the screen if not stored in a variable:

```
sherpa> 36/6
6
sherpa> sqrt(2.3*4.5)
3.21714
sherpa> y = sqrt(2.3*4.5)
sherpa> 2*y
6.43428
```

Loading S−Lang code

To try out these examples, save the code to a file − test.sl for example − and then load into ChIPS or Sherpa using:

```
unix% chips --slscript test.sl
```

or

```
  chips> () = evalfile("test.sl")
```

The functions will then be available for use (in the examples below we use the evalfile() method). Note that you can redefine the same function within a running S−Lang (ie Sherpa or ChIPS) session, so a second evalfile() call will pick up any changes made to the file.

S−Lang code versus Sherpa/ChIPS scripts

Sherpa and ChIPS can both run scripts, that is files containing a list of commands that you could also enter interactively, using

```
  unix% chips test.chp
```

for a ChIPS script and

```
  unix% sherpa test.shp
```

for a Sherpa script. Simple S−Lang commands can be used in either type of script, however multi−line statements − such as function definitions and loops − are not allowed. To use these, you must write a S−Lang script, as in the examples below. These S−Lang scripts can call Sherpa and ChIPS commands, but only through the sherpa_eval() and chips_eval() commands (see their respective ahelp documents for further information). See examples 8 and 11 for a demonstration of using ChIPS commands in a S−Lang script.

The examples are listed by number, together with a brief description of the main S−Lang feature they introduce. Below we list these titles, and briefly describe the code, which can be found in the examples section.

1 − Defining a function

Here we show how you write a S−Lang function, using as an example the output() function to print a string to the screen.

2 − Handling a variable number of arguments

The voutput() function is defined to extend the functionality of output() by allowing a variable number of arguments.

3 − Manipulating structures

The Varmm library input/output routines make use of S−Lang structures. We describe a function − print_struct() − that can print out the fields and values in a structure, and compare the results to the Varmm print() function.

4 − Looping through arrays

The function calc_sum() is described, which will loop through a one−dimensional array, calculating the sum of all the elements. We show how the use of the foreach loop makes the code faster, and able to cope with multi−dimensional arrays.

5 − Returning more than one variable

A minmax() function is described which will return more than one variable to the caller. As with the previous example, several versions of the code are presented. This function also introduces an alternative way of specifying the parameters to a function by reading them from the stack.

6 − Putting it all together (calculating statistics of an array)

The calc_stats() routine pulls together many of the previous examples to return a number of simple statistics for the supplied array. To simplify the code, it uses the S−Lang _reshape() function to handle multi−dimensional arrays.

7 − Returning values using a structure

With a few simple changes, calc_stats() from example 7 can be made to return a structure containing all the statistics, rather than a long list of values.

8 − Using ChIPS from S−Lang

The chips_eval() command is used to create a ChIPS plot − here the RATE and TIME columns of a file created by the lightcurve tool. We take advantage of the curve() function to plot the data, and introduce the label_plot() functions as one way to write S−Lang "wrappers" around ChIPS commands.

9 − Selecting subsets of arrays

A simple function, threshold(), is used to illustrate some of the powerful array−manipulation capabilities of S−Lang. Here we find all elements in an array which exceed a given value.

10 − Handling variable numbers and types of function arguments

A number of the examples have dealt with the different ways of handling function arguments. The check_cols() function shows how a S−Lang function can be written to accept a variable number of arguments, and handle different types of variables for some of the arguments.

11 − Using associative arrays

Associative arrays use strings, rather than numbers, to index array elements (they are similar to hashes in perl). We use such an array to allow a routine to accept a set of ChIPS plot options (see the description of the ChIPS state object in "ahelp chips") to change the output of a ChIPS plot.

12 − Using the ChIPS state object

This is a rewrite of example 11, using the ChIPS state object rather than an associative array for setting the plot attributes. We also show how you can clear the stack of arguments if the function has been called incorrectly and you wish to return to the calling code, rather than stop with an error.

Notes

ChIPS is used in the examples below, although they should all work within Sherpa too. Please note that these examples are intended to introduce certain aspects of S−Lang and its use within CIAO tools. They are not necessarily the best, or only, way to perform the given task! As with C, S−Lang does not enforce a particular formatting style; the examples below are written so as to be readable in an eighty−column wide terminal window. Text after a % (unless it is in a string) is a comment.

Example 1

Defining a function

```
  % output( string )
  % − prints string to stdout, and then prints a new line
  %
  define output (x) {
    () = fputs (x, stdout);
    () = fputs ("\n", stdout);
  }
```

Here we define a function − output() − that provides an alternative to the S−Lang output routines such as message() and fputs() [see "ahelp message" and "ahelp fputs" for information on these functions]. It takes a single input, here labelled x, and uses fputs() to print it to stdout, which is generally going to be the screen. The second fputs() line prints out a newline character, so that you do not have to add it to the string yourself. The first three lines − ie those beginning with a % character − are comments.

```
  chips> () = evalfile("example1.sl")
  chips> x = "a string"
  chips> output(x)
  a string
  chips> output("and another string")
  and another string
```

Although there is no mention of a variable type in the function, it will only work with strings − as shown below − since fputs() requires a variable of type String_Type as its first parameter. So, if you wish to print out a number you need to convert it to a string, as in the second call to output below.

```
  chips> output(23)
  Type Mismatch: Unable to typecast Integer_Type to String_Type
  Type Mismatch: output(23);
  chips> output(string(23))
  23
```

The Varmm print() function (see "ahelp varmm print") avoids the need for such type conversion.

Example 2

Handling a variable number of arguments

As the previous example is only good for a single string, we define another function − voutput() − that will take a number of arguments, together with a C−style formating statement (see "man printf"), and then send the resulting string to output() to print out.

```
  define voutput () {
    variable args = __pop_args (_NARGS);
    output (sprintf (__push_args (args)));
  }
```

In contrast to output(), voutput() can take a variable number of arguments, so no variable names are given in the () after the function name. Instead the __pop_args() and __push_args() functions are used to manipulate the stack and so access these variables (the number of arguments passed to the function is stored in the _NARGS variable). The Functions section of the "A Guide to S−Lang Language" at http://www.s−lang.org/doc/html/slang−9.html contains more information of how to write functions, and examples 5 and 10 show alternative ways to read function parameters from the stack.

Here we have the output() and voutput() code in two different files, although in normal use you would probably have them in the same file (the output function must be defined before voutput).

```
chips> () = evalfile("example1.sl")
chips> () = evalfile("example2.sl")
chips> voutput("Number = %d", 23)
Number = 23
chips> x = 0.1
chips> y = sin(x)
chips> voutput("The sine of %3.1f\nis %f", x, y )
The sine of 0.1
is 0.099833
```

Example 3

Manipulating structures

The output of the Varmm readfile() family of functions ("ahelp readfile") is a S−Lang structure that contains the contents of the input file. The following example − taken from the S−Lang Intrinsic Function Reference guide at http://www.s−lang.org/doc/html/slangfun.html − shows how such structures can be examined and manipulated, without having to know the actual field names. Example 7 shows how structures can be created.

```
define print_struct (s) {
  variable name, value;

  message( "Field\tValue" );
  foreach (get_struct_field_names (s))
  {
    name = ();
    value = get_struct_field (s, name);
    vmessage (" %s\t%s", name, string(value));
  }
}
```

Given a structure, print_struct() loops through each field and prints out both the name and the contents (the "\t" prints a tab character). The syntax of the foreach statement is discussed in section 8 of the S−Lang guide (http://www.s−lang.org/doc/html/slang−8.html); it is used to loop through the elements of a "container object" − here the fields in a structure. At each iteration, the name of the current field is pushed onto the stack, from where it is placed into the variable called name. Once the current field name is known, its value can be extracted and then printed out to the screen. We have used the message() and vmessage() functions of S−Lang rather than those just defined above.

```
chips> () = evalfile("example3.sl")
chips> img = readfile("img.fits")
chips> print_struct(img)
Field   Value
 _filename      img.fits
 _path  /data/ciao/
 _filter        NULL
 _filetype      11
 _header        String_Type[268]
 _transform     TAN
 _naxes 2
 pixels Float_Type[1024,1024]
 min    Double_Type[2]
```

```
  max    Double_Type[2]
  step   Double_Type[2]
  crval  Double_Type[2]
  crpix  Double_Type[2]
  crdelt Double_Type[2]
```

The output of print_struct() can be compared to that of the print() function provided by the Varmm library, which automatically handles structures (see "ahelp varmm print"):

```
chips> print(img)
_filename        =  img.fits
_path            =  /data/ciao/
_filter          =  NULL
_filetype        =  11
_header          =  String_Type[268]
_transform       =  TAN
_naxes           =  2
pixels           =  Float_Type[1024,1024]
min              =  Double_Type[2]
max              =  Double_Type[2]
step             =  Double_Type[2]
crval            =  Double_Type[2]
crpix            =  Double_Type[2]
crdelt           =  Double_Type[2]
```

Example 4

Looping through arrays

In this example we show how you can loop through an input array, here calculating the sum of all the elements in a 1−D array.

```
% calc_sum() computes the sum of elements of a 1−D array.
%
define calc_sum (x)
{
  variable num = length(x);
  variable total, i;

  total = x[0];
  % start looping at the second element
  for ( i = 1; i < num; i++ )
  {
    total += x[i];
  }

  return total;
}
```

The length() commands returns the number of items in the input array, here x (note that S−Lang − as in C − starts arrays at an index of 0). To loop through each element we use the for() statement, which follows the C syntax. Once the loop has finished − ie i equals num − the loop stops and we return the value of total.

Once loaded, the calc_sum() function can be used as any other S−Lang function:

```
chips> () = evalfile("example4.sl")
chips> x = [2:10:0.5]
chips> y = sin(x)
chips> print(calc_sum(y))
```

```
1.55481
```

Note that calc_sum() is restricted to 1D arrays since we have to access each element by its array index. However, the foreach command − discussed in example 5 − can be used to loop over all elements in an array, whatever the dimensionality and is faster than using for(). The S−Lang run−time library already contains the sum() function (see "ahelp sum"), which is essentially the same as the following:

```
% this is faster than the previous version, and will handle
% numerical arrays of any dimensionality
%
define calc_sum (x)
{
  variable total = 0;
  foreach ( x ) { total += (); }
  return total;
}
```

Example 6 shows another way of handling multi−dimensional arrays.

Example 5

Returning more than one variable

In this function, we loop through an array and return the the minimum and maximum values stored in the array. Unlike previous examples, this routine explicitly reads the array from the stack (the "variable a = ();" line) instead of declaring it in the brackets after the function name.

```
define minmax() {
  variable a = (); % get parameter value from the stack

  variable i, min, max;
  min = a[0]; % NOTE: assuming that will not be sent an empty array
  max = a[0];
  for ( i = 1; i < length(a); i++ ) {
    if ( min > a[i] ) { min = a[i]; }
    if ( max < a[i] ) { max = a[i]; }
  }

  return ( min, max );
}
```

The routine finishes by returning both the minimum and maximum values on the stack, which can be picked up by the caller:

```
chips> () = evalfile("example5.sl")
chips> x = [2:10:0.5]
chips> y = sin(x)
chips> ( minval, maxval ) = minmax(y)
chips> vmessage("min = %f max = %f", minval, maxval )
min = −0.977530 max = 0.989358
```

A faster version of the code uses foreach to place the current array element on the stack, rather than the C−style for command used above:

```
define minmax() {
  variable a = ();

  variable val, min, max;
```

```
    min = a[0];
    max = a[0];
    foreach ( a ) {
      val = ();
      if ( min > val ) { min = val; }
      if ( max < val ) { max = val; }
    }

    return ( min, max );
  }
```

An alternative to looping through the arrays is to use the min() and max() functions (see "ahelp min" and "ahelp max"). Although this will be slower than the previous version − since it requires 2 loops through the array − it will work for arrays of any dimensionality, whereas the previous versions are for 1D arrays only.

```
  define minmax() {
    variable a = ();
    return ( min(a), max(a) );
  }
```

Example 6

Putting it all together (calculating statistics of an array)

Here we show a much longer routine that calculates a number of statistics for an array. As with the minmax() function, the input array is read from the stack rather than declaring it after the function name. The routine begins by performing two checks on the input: was a value sent in, and if so, is it an array?

If these checks are passed, we find out if the array has more than one dimension. If it does then we use the _reshape() function to create a one−dimensional copy of it. This simplifies the code at the expense of memory, since a copy of the input array will be created for multi−dimensional arrays.

The remainder of the routine calculates the various statistics: the mean, min, max and total on the first pass through the array, followed by the calculation of the root−mean square value, and finally a simple − but slow − way to calculate the median value using the S−Lang array_sort() function.

```
  % Usage:
  %   ( mean, rms, median, min, max, total, ngood )
  %     = calc_stats( array_var )
  %
  % Aim:
  %  calculate simple stats for the supplied array
  %  array_var can be of any dimensionality - all
  %  elements are used in the calculation. If the
  %  array is > 1D, then a 1D copy of the array is
  %  made, which means the routine can use a lot
  %  of memory if sent a large array
  %
  %  The routine includes code to calculate the median of
  %  the array, but it's a simple, and so slow, method.
  %
  define calc_stats() {
    % Have we been called correctly?
    if (_NARGS != 1)
      usage( "() = calc_stats(a)\nwhere a is an array of numbers" );
    variable a = ();
    if (typeof(a) != Array_Type)
```

```
      usage( "() = calc_stats(a)\nwhere a is an array of numbers" );

    % total number of elements in array (whatever the dimensionality)
    variable npts = length(a);

    % Find out the size of the array: reshape if not 1D
    % (we are not interested in array dimensions or type)
    variable n1;
    ( , n1, ) = array_info( a );
    if ( n1 > 1 )
      a = _reshape( a, [npts] ); % make a copy

    % calculate the stats
    variable total, minval, maxval, value;

    total  = 0;
    minval = a[0];
    maxval = a[0];

    % loop through array elements
    foreach (a) {
      value = ();
      total += value;
      if ( value < minval )      { minval = value; }
      else if ( value > maxval ) { maxval = value; }
    }

    % ensure we have a floating−point calculation
    variable mean = total * 1.0 / npts;

    % calculate the RMS using the corrected two−pass algorithm
    variable diff, sum, sumsq;
    sum   = 0.0;
    sumsq = 0.0;
    foreach ( a ) {
      % diff = current element − mean
      diff   = () − mean;
      sum   += diff;
      sumsq += (diff*diff);
    }

    % again ensure we are doing this in floating point
    variable rms = sqrt( (sumsq − sum*sum*1.0/npts) / (npts−1.0) );

    % calculate the median using a rather simple/slow algorithm
    % − for the median of an even−sized array we take the average of
    %   the 'low' and 'high' values
    variable index = array_sort( a );
    variable midpt, median;
    midpt = (npts−1) / 2;
    if ( int(midpt) == midpt ) {
      % npts is odd
      median = a[index[midpt]];
    } else {
      % npts is even
      median = 0.5 * ( a[index[midpt−0.5]] + a[index[midpt+0.5]] );
    }
    return ( mean, median, rms, minval, maxval, total, npts );

  } % calc_stats()
```

In the following, we use calc_stats() to calculate the mean, median, rms, minimum and maximum values of a one−dimensional array. Note that we ignore the last two items returned by calc_stats(), ie the sum of the input array and the number of elements.

```
chips> () = evalfile("example6.sl")
chips> x1 = [2:10:0.5]
chips> y1 = sin(x1)
chips> (mean1,median1,rms1,minv1,maxv1,,) = calc_stats(y1)
chips> vmessage("mean=%f median=%f rms=%f",mean1,median1,rms1)
mean=0.097176 median=0.141120 rms=0.700011
```

Since calc_stats() can handle multi−dimensional input arrays, in the following we create an array that contains the same values as used previously, but stored in a two−dimensional format.

```
chips> x2 = [ [2:6:0.5], [6:10:0.5] ]
chips> y2 = sin(x2)
chips> (mean2,median2,rms2,minv2,maxv2,,) = calc_stats(y2)
chips> vmessage("mean=%f median=%f rms=%f",mean2,median2,rms2)
mean=0.097176 median=0.141120 rms=0.700011
```

Example 7

Returning values using a structure

An alternative method of returning multiple parameters from a function is to use a structure. For example, the "return" statement from the calc_stats() function in example 6 can be changed to:

```
variable stats = struct { mean, median, rms, minval, maxval,
        total, npts };
set_struct_fields( stats, mean, median, rms, minval, maxval,
        total, npts );
return stats;
```

which changes the usage of calc_stats() to

```
chips> () = evalfile("example7.sl")
chips> x2 = [ [2:6:0.5], [6:10:0.5] ]
chips> y2 = sin(x2)
chips> s = calc_stats(y2)
chips> print(s)
mean             =  0.0971758
median           =  0.14112
rms              =  0.700011
minval           =  −0.97753
maxval           =  0.989358
total            =  1.55481
npts             =  16
chips> vmessage( "npts = %d", s.npts )
npts = 16
```

Example 8

Using ChIPS from S−Lang

The following set of routines show how you can plot the TIME and RATE column of a file created by the lightcurve tool. See "ahelp chips" for more information on interfacing S−Lang and ChIPS code.

```
%
% label_plot( xlabel, ylabel, title )
% adds the three strings as labels to the current plot
%
define label_plot(xlabel,ylabel,title) {
  () = chips_eval( "xlabel '" + xlabel + "'" );
  () = chips_eval( "ylabel '" + ylabel + "'" );
  () = chips_eval( "title  '" + title + "'" );

} % label_plot()

% lcurve("lcurve.fits") plots up the TIME and RATE
% columns of the file lcurve.fits
%
define lcurve (filename) {
  % read in data − check we got some
  variable lc = readfile(filename);
  if ( lc == NULL )
    error( "lcurve() unable to read data from " + filename );

  % assume the TIME and RATE fields exist
  % we could check for this using e.g. get_struct_field_names()
  variable time = get_struct_field( lc, "TIME" );
  variable rate = get_struct_field( lc, "RATE" );

  % use the filename as the title for the header − protect any
  % '_' in it or else the next character will become a subscript
  variable title;
  ( title, ) = strreplace( filename, "_", "\\_", strlen(filename) );

  % plot the curve − set redraw to off to stop flickering
  () = chips_auto_redraw( 0 );
  chips_clear();
  () = curve( time, rate );
  () = chips_eval( "step" ); % plot as a "histogram"
  () = chips_eval( "symbol none" );
  label_plot( "Time (s)", "Rate (/s)",
        "Lightcurve for: " + title );
  () = chips_auto_redraw( 1 );

} % lcurve()
```

If lc.fits is the output from lightcurve, then "lcurve(lc.fits)" will plot the rate against time, and change several of the plot options from their default values. The section on the ChIPS state object in "ahelp chips" describes another way of changing the plot defaults from S−Lang and example 11 presents one way to do this in S−Lang.

Example 9

Selecting subsets of arrays

S−Lang has powerful array−manipulation capabilities, as discussed in Section 11 of the "S−Lang Guide" at http://www.s−lang.org/doc/html/slang−11.html. The "[[]]" syntax can be used to access subsets of an array:

```
chips> x = Integer_Type [2,5]
chips> print(x)
0       0       0       0       0
0       0       0       0       0
chips> x[0,*] = [0:4]
chips> x[1,*] = [5:9]
chips> print(x)
0       1       2       3       4
5       6       7       8       9
chips> y = x[1,[2:3]]
chips> print(y)
7
8
```

Here we created a two−dimensional array, x, and then extracted the third and fourth elements of the second row, storing the result in the variable y. An alternative way to access array elements is the where() function, which returns an array containing the index of those elements which match a given condition. We illustrate this concept with a function that returns only those elements in a pair of arrays for which the value of the first array is greater than the supplied threshold:

```
define threshold (x,y,thresh) {
  % find the position of those elements in x which are > thresh
  % (assuming x and y are 1D and the same size)
  %
  variable i = where( x > thresh );

  % if there are no elements, then return two NULL values,
  % otherwise return the requested elements
  %
  if ( length(i) == 0 )
    return ( NULL, NULL );
  else
    return( x[i], y[i] );
} % threshold
```

In the example below, we use this routine to find out which x values have a sine that is larger than 0.2. In order to print out the values of both of the new arrays, we use the writeascii() Varmm function ("ahelp writeascii").

```
chips> () = evalfile("example9.sl")
chips> x = [−10:10]
chips> y = sin(x)
chips> (ynew,xnew) = threshold(y,x,0.2)
chips> writeascii( stdout, xnew, ynew )
−10     0.544021
−6      0.279415
−5      0.958924
−4      0.756802
1       0.841471
2       0.909297
7       0.656987
8       0.989358
9       0.412118
```

Example 10

Handling variable numbers and types of function arguments

Example 1 shows how you can use the function declaration to specify the name of function arguments, and example 2 demonstrates one way of handling a variable number of arguments. The following code shows how one can deal with a function that accepts a range of parameters (both in number and type). If the incorrect number of parameters is sent in, we exit with a message giving the correct calling sequence. The same message is printed if the type of the first two arguments are not correct.

```
%
% check_cols( a, b, colname, scale [, tol ] )
%
% check that two columns are the same, apart from a scaling factor,
% to within a given tolerance (defaults to 0)
%
% ie fail if any element of
%    abs( a.<colname> * scale − b.<colname> )
% is greater than tol
%
% a and b can be either a structure, containing the given column
% as a field (ie the return value of a varmm readfile() command) or
% the name of a file which contains this column
%
define check_cols () {

  variable usage_str =
    "check_cols(file1|struct1,file2|struct2,colname,scale[,tol])";

  % see if correct number of arguments were supplied
  variable in1, in2, colname, scale, tol;
  if ( _NARGS == 5 ) {
    ( in1, in2, colname, scale, tol ) = ();
  } else if ( _NARGS == 4 ) {
    ( in1, in2, colname, scale ) = ();
    tol = 0;
  } else {
    usage( usage_str );
  }

  % The first two parameters can be either a filename (String_Type)
  % or a structure (Struct_Type). If the former then we need
  % to read in the data from the file
  %
  % if we have a structure, then do nothing, otherwise read in
  % the column from the specified file, using the data model
  % to make sure we only get the required column
  %
  variable cols1, cols2;

  % first file/structure
  if ( typeof(in1) == Struct_Type ) {
    cols1 = in1;
  } else if ( typeof(in1) == String_Type ) {
    cols1 = readfile( in1 + "[cols " + colname + "]" );
    if ( cols1 == NULL )
      error( "Unable to read the " + colname + " column from "
        + in1 );
  } else usage( usage_str );

  % second file/structure
  if ( typeof(in2) == Struct_Type ) {
    cols2 = in2;
  } else if ( typeof(in2) == String_Type ) {
    cols2 = readfile( in2 + "[cols " + colname + "]" );
    if ( cols2 == NULL )
      error( "Unable to read the " + colname + " column from "
        + in2 );
  } else usage( usage_str );

  % get references to the columns (assume fields exist)
  variable c1 = get_struct_field( cols1, colname );
```

```
    variable c2 = get_struct_field( cols2, colname );

    % is the absolute difference > the tolerance anywhere?
    variable absdiff = abs( scale * c1 − c2 );
    variable index = where( absdiff > tol );

    % if there are any elements in index, then the 2 columns do
    % not match to within the tolerance
    if ( length(index) != 0 ) {
      % find out the max value
      variable maxval = max( absdiff[index] );

      error( "2 columns do not agree within a tolerance of " +
        string(tol) + "\n  max diff = " + string(maxval) );
    }

  } % check_cols()
```

The "@" − or dereference − operator used below creates a copy of a structure, as described in chapter 13 of the "S−Lang Guide" at http://www.s−lang.org/doc/html/slang−13.html.

```
  chips> () = evalfile("example10.sl")
  chips> x = struct { ciao }
  chips> x.ciao = [ 1, 2, 3, 4.5, 9.638 ]
  chips> y = @x  % y is a copy of x
  chips> z = @x  % so is z, until we change the values (next line)
  chips> z.ciao = [ 1, 2, 3, 4.501, 9.368 ]
  chips> writeascii(stdout,x.ciao,y.ciao,z.ciao)
  1       1       1
  2       2       2
  3       3       3
  4.5     4.5     4.501
  9.638   9.638   9.368
  chips> check_cols( x, y, "ciao", 1 )
  chips> check_cols( x, z, "ciao", 1, 0.01 )
  2 columns do not agree within a tolerance of 0.01
    max diff = 0.27
  check_cols( x, z, "ciao", 1, 0.01 );
```

Note that CIAO 3.0 includes the dmdiff tool which allows you to compare tables for equality.

Example 11

Using associative arrays

S−Lang has a type of array called an associative array that uses strings, rather than integers, as array indices (they are similar to hashes in Perl). For instance, in the following we define opt as being an associative array that contains Integer_Type values (note that the keys are always String_Type). We then set and access an element of the array using as the index the string "linestyle" (the allowed values of _chips are discussed in "ahelp slang−chips").

```
  chips> opt = Assoc_Type [Integer_Type]
  chips> opt["linestyle"] = _chips->dotdash
  chips> key = "linestyle"
  chips> print(opt[key])
  2
```

The following example shows how you can use such an array to send a function a set of options, in this case to control the format of a plot.

```
%
% sl_plot ( x, y [, options ] )
%
% plot a curve (x and y arrays), using the ChIPS state object
% to control the appearance (the optional options associative
% array)
%
% The allowed keys for the options associative array are the
% same as the ChIPS state object (see "ahelp chips"), ie:
%   curvestyle symbolstyle linestyle etc
%

% create the list of allowed options fron the ChIPS state object
% − only the keys are important here, the values are not
%
variable _chips_opts = Assoc_Type [Integer_Type];
foreach ( get_struct_field_names(chips) ) {
  variable nval = ();
  _chips_opts[nval] = 0;
}

define sl_plot () {

  variable usage_str = "sl_plot( x, y [, options ] )\n"
      + "where x and y are arrays and options is an "
      + "associative array\n";
  variable useropts, origopts, x, y;

  % check usage − use the switch() structure instead of a set
  % of "if/else if" commands.
  %
  switch ( _NARGS )
    { case 3 :
        ( x, y, useropts ) = ();
        % all args removed from stack, so do not need to clear it
        % if there is an error
        if ( typeof( useropts ) != Assoc_Type )
          usage( usage_str ); }
    { case 2 :
        useropts = NULL;
        ( x, y ) = (); }
    { % incorrect number of arguments
      usage( usage_str ); }

  % if useropts is non−NULL, then update the ChIPS state object
  % storing the original values in origopts
  %
  if ( useropts != NULL ) {
    % we do not define a type for origopts as we have to be able to
    % store integers (eg curvestyle) and strings (eg tmpdir)
    origopts = Assoc_Type [];

    % loop through the input options
    variable key;
    foreach ( assoc_get_keys( useropts ) ) {
      key = ();

      % is this a valid key?
      if ( assoc_key_exists( _chips_opts, key ) ) {

        % store original value, update ChIPS state object
        origopts[key] = get_struct_field( chips, key );
```

```
          set_struct_field( chips, key, useropts[key] );

        } else {
          vmessage( "Ignoring unknown ChIPS option %s", key );
        }

      } % foreach
    } % if: useropts != NULL

    % plot the data
    %
    chips_clear();
    () = curve( x, y );

    % restore the original ChIPS state object (if we changed it)
    %
    if ( useropts != NULL ) {
      foreach ( assoc_get_keys( origopts ) ) {
        key = ();
        set_struct_field( chips, key, origopts[key] );
      } % foreach
    } % if: useropts != NULL

    return;

  } % sl_plot()
```

If a third parameter is supplied to the function, it is checked to make sure it is of the correct type, with a "usage" message being printed if it isn't. Each key of the associative array is then checked to see if it is a valid option by checking to see if it is a key of the _chips_opts associative array. If it isn't, then we ignore it, otherwise we set the relevant field of the ChIPS state object to the new value. We store the old values of all those fields we change in the "origopts" associative array. This is then used after the plot has been created to restore the fields of the ChIPS object to its original state.

In the following example we show how you can use this function to control the look of a plot: calling sl_plot(x,y,opts) produces a plot in which the points are drawn as blue diamonds connected by a green curve. Note that although we explicitly set the ChIPS state object to have a symbol color of red, the value from the options array is used instead. As sl_plot() restores the original values of the ChIPS state object before returning, the final command produces a set of red crosses with no connecting curve.

```
  chips> () = evalfile("example11.sl")
  chips> x = [0:2*PI:0.1]
  chips> y = sin(x)
  chips> % set up the associative array
  chips> opts = Assoc_Type []
  chips> opts["curvestyle"] = _chips->simpleline
  chips> opts["symbolstyle"] = _chips->diamond
  chips> opts["curvecolor"] = _chips->green
  chips> opts["symbolcolor"] = _chips->blue
  chips> % set default plotting options
  chips> set_state_defaults("chips")
  chips> chips.symbolcolor = _chips->red
  chips> % create plot with desired attributes
  chips> sl_plot( x, y, opts )
  chips> % go back to using default attributes
  chips> % − plot in another "window"
  chips> chips_split( 1, 2, 2 )
  chips> () = curve(x,y)
```

Example 12

Using the ChIPS state object

Although the associative array is extremely useful, the code from example 11 can be simplified by taking advantage of the ChIPS state object and the Varmm set_state() command, as shown below. We also change the behaviour slightly, so that if the routine is called incorrectly, the routine returns rather than stops with an error message. In this case we have to make sure we have cleared the stack of any parameters sent in, which is done with the _pop_n() command.

```
%
% sl_plot ( x, y [, ChIPS state object ] )
%
% plot a curve (x and y arrays), using the ChIPS state object
% to control the appearance (the optional options associative
% array)
%
% unlike example 11, the optional third argument is a copy
% of the ChIPS state object, with its fields set to the desired
% plot attributes
%

define sl_plot () {

  variable usage_str = "sl_plot( x, y [, ChIPS state object ] )\n"
      + "where x and y are arrays";
  variable useropts, origopts, x, y;

  % check usage − use the switch() structure instead of a set
  % of "if/else if" commands.
  %
  switch ( _NARGS )
    { case 3 :
        ( x, y, useropts ) = ();
        % all args removed from stack, so do not need to clear it
        % if there is an error
        if ( typeof(useropts) != Struct_Type and
             typeof(useropts) != chips_State ) {
          print( usage_str );
          return;
        }
    }
    { case 2 :
        useropts = NULL;
        ( x, y ) = ();
    }
    { % incorrect number of arguments, remove any arguments from
      % the stack
      %
      if ( _NARGS != 0 ) _pop_n( _NARGS );
      print( usage_str );
      return;
    }

  % if useropts is non−NULL, then update the ChIPS state object
  % storing the original values in origopts
  %
  if ( useropts != NULL ) {
      origopts = @chips;
      set_state( "chips", useropts );

    } % if: useropts != NULL

    % plot the data − use the S−Lang curve() function since we
    % do not need the full functionality of the ChIPS curve command
    %
    chips_clear;
    () = curve( x, y );

    % restore the original ChIPS state object (if we changed it)
    %
    if ( useropts != NULL ) set_state( "chips", origopts );

    return;

  } % sl_plot()
```

This function is much simpler than that of example 11 because the set_state() function ensures that only valid attributes are copied across to the state object. It is used in a similar manner to the previous version:

```
  chips> () = evalfile("example12.sl")
  chips> x = [0:2*PI:0.1]
  chips> y = sin(x)
  chips> % set up the copy of the plot options
  chips> opts = @chips
  chips> opts.curvestyle = _chips−>simpleline
  chips> opts.symbolstyle = _chips−>diamond
  chips> opts.curvecolor = _chips−>green
  chips> opts.symbolcolor = _chips−>blue
  chips> % set default plotting options
  chips> set_state_defaults("chips")
  chips> chips.symbolcolor = _chips−>red
  chips> % create plot with desired attributes
  chips> sl_plot( x, y, opts )
  chips> % go back to using default attributes
  chips> % − plot in another "window"
  chips> chips_split(1,2,2)
  chips> () = curve(x,y)
```

See Also

calibration
    caldb
chandra
    coords, guide, isis, level, pileup, times
chips
    chips, chips_eval
concept
    autoname, parameter, stack, subspace
dm
    dm, dmbinning, dmcols, dmfiltering, dmimages, dmimfiltering, dmintro, dmopt, dmregions, dmsyntax
gui
    gui
modules
    paramio, pixlib, stackio, varmm
sherpa

sherpa_eval

slang

math, overview, slang, variables

tools

ascii2fits

URL:
http://cxc.harvard.edu/ciao3.4/slang−tips.html
Last modified: December 2006