Brannigan Lab Guide: TCL Scripting in VMD

ver 1.0

Liam Sharp, Grace Brannigan

October 1, 2020

Contents

1 Section 1: Basics

The introduction to TCL on the VMD website tutorial can be found here, `https://www.ks.uiuc.edu/Training/Tutorials/vmd/tutorial-html/node4.html`.

To Do: Work through the tutorial.

The tutorial does skip an important aspect of programming: if...else statements. In programming, we use if..else statements to consider conditions in our scripts. TCL is no different. Below is a syntax comparison for how python and TCL would expect an if statement:

Python:

```
if 1 > 10:
    print("This statement makes no sense!")
elif i < 0:
     print("This statement makes no sense either!")
else:
    print("This statement is true!")
```

TCL

```
if {1 > 10} {
    puts "This statement makes no sense!"
} elseif {1 < 0} {
    puts "This statement makes no sense either!"
} else {
    puts "This statement is true!"
}
```

Write a script to track three water beads over 100 frames. Using memb_d_01 (found in the slack channel where you found this guide) with resid's 5636, 42609, and 64952. For this script please include:

- An error check (confirm the bead is water, confirming the correct resid is used,...)
- An output file for each of the water beads (frame, position)
- Documentation using ;# explaining steps of your code, ;# tends to be more stable than #

2 Section 2: Lists

2.1 General List Information

Lists are a corner stone of TCL. If you would use a 1D array in another programming language, it probably makes sense to use a list in TCL. However, common usage is slightly different, because while lists are easy to build and easy to print, replacing elements is much more unwieldy.

You can **create a list** using the *list* command inside square brackets, or directly using curly brackets or quotation marks:

```
set var1 [list item1 item2 item3 item4 .... itemn]
set var2 {item1 item2 item3 item4 .... itemn}
set var3 ``$x1 $x2 $x3''
```

Lists are just stored as strings, and they can store anything that could be stored in a string, including numbers, names of other variables, or even other lists. (You can output your whole list using puts.) While in most programming languages it would be very silly to store several numbers inside a string, TCL has some more advanced string parsing commands that make this work. Further, puts is also list compatible, and will output all contents of the list as a wrapped line.

One of the most commonly used list commands is the *lappend* function, which will **add an indexable value** to the end of an existing list:

```
set str "Hi there!"
set var {}
lappend var 1
lappend var "a"
lappend var $str
puts $var
```

Note that you would not put a $ in front of the list name in *lappend*. This is because *lappend* is changing the list, not just accessing it.

The *lappend* command is commonly used because lists are really strings, not arrays, so it is much easier to concatenate lists than replace individual elements of lists. For instance, if you were trying to fill an array in C or python, you might declare an array, loop through every index in the array, and assign a value, like so:

```
float values[5];
for(int i = 0; i < 5; ++i) {
        values[i]= 2*i
}
```

But in TCL, we would do this instead:

```
set values {}
for {set i  0} {$i < 5} {incr i) {
        lappend values [expr 2.0*$i]
}
```

To **determine the length** of the list, use the *llength* command:

```
llength $var
```

and to set another variable to that length, just put it in square brackets:

```
set listLength [llength $var]
```

While replacing a list element is a little tricky, accessing it is straightforward. To **access** any element in a list, using the *lindex* command. For instance,

```
lindex $var 0
```

returns the first element of the list. Note that here we did have a $ in front of var because we are accessing it but not changing it.

If you build a list using *lappend*, the list will remain in the order you built it, which might not be the order you want. To **sort** a list, use the *lsort* command. For example, to put your list in descending order (-decreasing).

```
lsort -decreasing $var ;#descending order
```

while the -increasing flag can be used for ascending order. If you need unique values in your list, use the -unique flag.

```
lsort -unique $var ;# unique values only
lsort -decreasing -unique $var ;# unique values in a descending order
```

For a full list of possible sorting flags, see `https://www.tcl.tk/man/tcl8.0/TclCmd/lsort.htm`

2.2 List Manipulation:Foreach Loops

General for loops were covered in the TCL/VMD tutorial mentioned in Section 1. If you wanted to access each element in a list, you could loop through the indices using *for* and then use *lindex* to call the corresponding element:

```
for {set i 0} {$i < [llength $var]} {incr i} {
   puts "[lindex $var $i]"
}
```

Since this type of loop through lists is so common, TCL provides a shortcut. *Foreach* combines a for loop and an lindex in one command, which allows us to write the previous loop like so:

```
foreach v $var {
        puts $v
}
```

A foreach loop can also iterate over multiple variables at the same time, which allows us to treat a group of lists like columns in a 2D array:

```
foreach v1 $var1 v2 $var2 {
        puts ``$v1, $v2''
}
```

> Exercise:
> a) Make one list with the first names of everyone who comes to group meeting, and another list with our last names. Output each of our full names using only one foreach loop.
> b) Put both lists in alphabetical order using lsort and run the same foreach loop. (You should not end up with the right full names.)
> c) Output the correct full names, in alphabetical order by last name.

2.3 Examples of common list usage in VMD

1. Suppose you want to output the center of mass for residues 5 in segname PROA, 4 in segname PROB, 10 in segname PROC, 12 in segname PROD, 6 in segname PROE. You could use the list commands covered here, like so:

```
set segList [list PROA PROB PROC PROD PROE]
set  resList [list 5 4 10 12 6]
set comList {}
foreach segname  $segList res $resList {
        set sel [atomselect  top ``resid $res and segname $segname'']
        lappend comList [measure center $sel]
        $sel delete
}
puts $comList
```

We first defined a list containing the particular values we wanted, because it allows you to iterate through them, keep the code tidy, and change them easily. resList is technically a separate list, but we iterated through them in parallel, effectively treating our two lists as two columns of a 5x2 array. Lastly, always delete selections once you are done with them.

2. Suppose you want to swap every residue in segname PROA to be in segname PROB, and vice versa:

```
set atomSelA [atomselect top ''segname PROA'']
set segA [$atomSelA get segname]

set atomSelB [atomselect top ''segname PROB'']
set segB [$atomSelB get segname]

$atomSelA set segname $segB
$atomSelB set segname $segA
$atomSelA delete
$atomSelB delete
```

Note that since atomSelA contains multiple atoms, when you use "$atomListA get" it returns a list with one value of whatever you were getting for each atom. So segA is also a list, but here it is just PROA over and over again, because every atom in the selection has the same segname. Also, "$atomListA set" is vectorizeable : we are changing all the atoms simultaneously by passing a list of values and a list of atoms.

3. Suppose you want to output the **unique** resid of every atom within 5A of protein amino acids:

```
set sel [atomselect top ''within 5 of protein'']
set resList [$sel get resid]
set uniqueList [lsort -unique $resList]
$sel delete
puts $uniqueList
```

To Do: Load covE.psf and covE.pdb in this channel, and try doing each of the example tasks above yourself. You can consult the examples and even copy directly from them, but you'll know you are fluent once you can truly do them on your own.

2.4 List Manipulation:List Math

Exercise: Define two lists, each containing 6 numbers of your choice. Using a foreach loop, generate two new lists, one with the element-wise sum and one with the element-wise difference of the two original lists.

In general, when the operations carried out within the loop don't depend on the previous iteration, they should be *vectorizeable*: the computer should be able to do them simultaneously. The example above is one such case: element 2 of the summed vector only depends on element 2 of each individual vector, not element 1 or 0.

TCL is not, itself, vectorizeable, but many VMD commands are. The most basic commands are simply vector operations, including vecadd, vecsub, vecscale, and the others listed here `https://www.ks.uiuc.edu/Research/vmd/current/ug/node193.html`

Exercise: Calculate the sum and difference lists from the previous problem without using any for loops

Exercise: Using the same system from section 1, construct a list containing the distance of bead 1000 from its position in the first frame. Each element in the list would correspond to a frame. You must have at least one error checking step and at least one use of either vecdist or vecsub & veclength. Output the list to a file after you have constructed it.

3 TCL Module Building and Testing

3.1 Procedures

When coding, it is important to maintain code readability using meaningful variable names, comments and general hygiene (ex: indentation, placement of comments, version control...). We have not discussed methods of coding how to run a process multiple times (double talk), when one time will work. The first step in preventing double talk is application of a loop, which we have previously seen. We can further extend this idea of double talk to procedures.

A procedure (proc) is the TCL equivalent of a python def. It takes in an argument, it can return data, is a reliable structure to preform a task over and over, and is essential to writing modular and reusable code.

Python:

```
def defName(arg1, arg2, ..., argN):
        Body of function
```

TCL:

```
proc procName {arg1 arg2 ... argN} {
        Body of proc
}
```

Example: You have a list defining the length of a side of various squares. You want to develop a script to take the area of the current index and compare (take the difference) of the current and previous square.
It may be tempting and convenient to build this as a single loop (and for the sake of prototyping that is true!):

```
set side [list 0 1 2 3 4] ;# unit cm
set previous 0 ;# stores previous area

foreach s $sid {
        set area [expr 1.0 * $s * $s]
        puts "Diff in area:"
        puts "[expr $area - $previous] cm^2"
        set previous $area
}
```

However, using a procedure (proc) will allow you to recall your area calculation, keep your code more manageable, and allow you to access function from other scripts.

```
proc Area_Square {side_in} {
        return [expr 1.0 * $side_in * $side_in]
}

proc Diff_Area {area1 area2} {
        return [expr 1.0 * $area1 - 1.0 * $area2]
}

set side [list 0 1 2 3 4] ;# unit cm
set previous 0 ;# stores previous area

foreach s $side {
        set area [Area_square $s]
        puts "Diff in area:"
        puts "[Diff_Area $area $previous] cm^2"
        set previous $area
}
```

A feature not always discussed but found in almost all proc/function building, is the default argument.

```
#Python
def foo (a, b=1):
        return a * b

#TCL
proc foo {a {b 1.0}} {
        return [expr $a * $b]
}
```

In python, the default argument can be defined declaring the variable and setting it equal to something. In TCL, the variable and what it is equal too are declared within braces. If you choose to use an argument instead of the default, you would call it as normal [foo 2 100] and the 100 would replace the 1 for variable b.

maybe a exersize on periodic boudnary conditions? That might be better to come later

> Exercize: You have recently read a paper that reported temperature constraint at 98.33 Fahrenheit. You want to repeat the experiments reported but need to convert the temperatures to Kelvin.
>
> - For each conversion (F to C to K), write a proc.
> - Debug your script with using the following tempuratutes (212F, 32F, 40F). Do the values in Celcius and Kevlin make sense?. They should be: 100C, 0C, 40C, 373.15 K, 273.15 K, and 313.15.

3.2 Associative Arrays

Associative Arrays should be treated with care. They are useful structures that behave more similarly to lists found in other programming languages (they can be indexed without need of lindex), and are thus an attractive option to use. They are slower and bulkier than a TCL list, have their own commands, and may slow down or complicate your code. Because of this, this document will only cover a small section of associative array usage, and strongly recommends using lists when ever possible.

An associative array can be instantiate in the same fashion as other variables, but you must define the "index" (names):

```
set arr(0) 1
set arr("Ted") "Bill"
set arr("X""Y") "Z"
```

The names for the array arr are: 0, "Ted", and "X""Y", they are not required to be 0 to n. If you need to determine the names of a given index:

```
array names arr ;# no $ in front of the variable
```

Calling $arr(name) will let you update, or puts out the information stored in it. If you want to print the entire array you need to use parray:

```
parray arr ;# no $ in front of the variable
```

Lastly, unlike python or matlab, or TCL lists, it can be challenging to return associative arrays from a proc. Instead is is closer to arrays or vectors in C and C++. I have found the most success using the upvar command.

Example: We want to return the area of a square to an associative array using a proc. We know the sides of the square are 1, 2, 3, 4, and 5 cm. To do this use the upvar command within the area calculating proc.

```
proc Area_Square {side arr} {
        upvar $arr a
        set a($side) [expr $side * $side]
}

set side [list 1 2 3 4 5]
set arr(0) 0

foreach s $side {
        Area_Square $s arr
}

parray arr
```

If you would like to know more on associative arrays please take a look at `https://www2.lib.uchicago.edu/keith/tcl-course/topics/arrays.html`

3.3 Testing

The nature of computational stem fields rely heavily on modularization and flexible, lightweight coding that can be used for a variety of procedures. Over time, as your analysis becomes more specific, old modules require updates and new methods. This can lead to bugs and inefficiency.

When building or updating a TCL proc, it is advised to:

- Write out what you want to do in English and pseudo code
- Test your ideas in the tk-console
- Write an ideal (usually hard coded) script and try to break it
- Ask youself do these results align what is expected?
- Convert script into a proc

But I have already written a TCL procedure. Should I still follow these steps to debug? Absolutely. Run each line of the procedure in the tk-Console, and confirm the lines produce expected results.

If the script has too many bugs, can I re-write it? It is ill advised to do this, though there are times this is the best idea. Rather than re-build a script form scratch, when you will likely make the same mistake again, learn what went wrong.

I think its just the computer making mistakes Lol. You should be a comedian. There is a possibility, but a program will do what you tell it to do. If it breaks, chances are its because you told it to do something you did not want it to do.

If my script is super slow, then can I re-write it? Good question! No. We can determine why a script is slowing down though. What part of the script is slow? Is it caused by a single procedure? Is it caused by an unnecessary command? We can test these questions with the time function.

Example: Using the time function, determine the average time it takes to run a for and a foreach loop and summing a list.
To call the time function

```
time {
    code you want to test
} iterations
```

The more iterations, the more accurate your run time is, but the more time it takes to calculate.

```
;# a list populated with 1 10000 times
set var [lrepeat 100000 1]
set out 0

;# Time how long it takes to iterate
;# and add using this list in a for loop

time {
    for {set i 0} {$i < [llength $var]} {incr i} {
        set out [expr [lindex $var 0] + $out]
    }
} 100

;# Time how long it takes to iterate
;#and add using this list in a foreach loop

set out 0
time {
    foreach v $var {
        set out [expr $v + $out]
    }
} 100
```

Exercise: Using the example from the Associative Array section, update the function to use a list instead of an array. Once this is debugged, test both versions with time { ... } 100, which runs faster?