SkinnyScript is an easy to learn, simple programming language. You have to install it on every computer that you want to run a skinny script on, but it is worth it. With skinny script, most basic tasks are easy. You can read, write, and append to files. You can get console input, as well as print to the console. You can generate a random number. You can, of course, compare two values with if. Goto allows you to do loops. In a later code example, we made a simple, very slow, md5 cracker. So, let's get started.

Chapter 1. Getting started

As you may notice, skinny script requires a file that contains a script. But it cannot be an RTF, it must be a text file. So, how do you create, edit, and save a text file. Well, there are a few applications that will do this. My favorite is MacVim. But, luckily for you, you have TextEdit on your mac. So, the way to make a text file is simple.

Step 1 there is an application on your computer called terminal. Open that.
Step 2 when the terminal window opens, type the following commands:

```
cd Desktop
touch textFile.txt
```

Step 3 Now you will have a text file on your desktop named textFile (.txt). If you open that with TextEdit (probably just double click), then you are now in a text document. Rename the text document, make copies, or do whatever. It is a shame that you cannot save a text file with textedit.
Step 4 Type code in the file, and save it. But, how do you write code? What should you put in the text file? That's where this document comes in. Let's write a little program that prints text to the console. Here is the code that we will put in TextEdit:

```
print [Hello world\n]
```

Ok, so, print is the function that prints text. Print takes 1 parameter, that is a string. A string is a piece of text. We tell it that we want to print "Hello World" by putting it in [and]. The "\n" at the end means "new line", which types enter basically. The reason that you can't just put Hello World\n without [and] then print will think that you're giving it another parameter at the space (" "). So, when passing a string, using [and] will do it. Ok, so, let's say we want to store data in memory and then print it out. Here's an example of that:

```
string $s
set $s [hello world]
print [$s\n]
```

In that example, we used two new functions, string, and set. Set assigns a value or variable to another variable. String just tells the computer that I want a string (piece of text) in memory that I can access by this name. The name is the second parameter. The $ represents a variable name. All variable names start with a $. The print command has [then $s then \n. It replaces $s with the contents of the variable. In this case, hello world. Then you can continue going on in the string for other characters. Here's another example of set and string:

```
string $1
string $2
string $3
```

```
set $1 [hello]
set $2 [world
set $3 $2
print [$1 $3\n]
```

As you can see there, I made three strings. One named $1, one $2, and one $3. I set $1 to be hello, I set $2 to be world, and I set $3 to be $2. Meaning that $3 is now world. So when I print $1 then space then $3 it will print “hello world”. Now, here’s the thing. Let’s say you don’t want a space after a variable name, for instance, say you want to print some text then a ‘.’ (period). The reason why you couldn’t do this is because it would think that ‘.’ was part of the variable name. So here is an example on how to do this:

```
string $1
set $1 [alex]
print [$1#.]
```

So the # tells skinny script “this is the end of the variable name”. Anywhere else that you use # it will be fine and not mean anything magic. Here’s an example of how to print there name, then a #, then a ‘.’.

```
string $name
set $name [alex]
print [$name##.]
```

The first # means something special, but the second doesn’t. You can randomly put a # in the thing and it is fine. But let’s say that you want to print a $ or a \ in the text. How would you do that? Well, here’s a script that does both:

```
print [\$ and \\ are hard to print]
```

So, as you can see, “\$” makes a $, and “\\” is one \. So, it’s the same for a “]” and a “[“. Here we go:

```
print [you do a string with \[ and \]]
```

Make sense to you? Hopefully it does.

Chapter 2. Variables

Let's go over some other data types. There are four main ones. Those are:

1. string
2. int
3. array
4. file

So, let's declare an int. Int is short for integer. Integer is a whole number. This is how we use one:

```
int $i
set $i (10)
print [$i\n]
```

That will print the number 10. When setting an int or making an int, you use (and) to represent it. So, when I do the set, I am setting it to 10. Simple equations can be done with this too. Here is an example

```
int $i
set $i (10 * 9)
print [$i\n]
```

The first thing to get is that the syntax for (and) is **very** specific. You cannot have a stray space, and you must either have one number, or a number, operation, then a second number. In that script, I used 10 * 9 (ten times nine). Now the int $i is set to 90, which will be printed out. The following int assignments will not work

```
int $i
set $i ( 10 * 9 )
set $i (10  *  9)
set $i (10 * 9 )
set $i (10 * 9 + 3)
set $i ((10 * 9) + 3)
```

None of those set commands will work properly. It will most likely just make $i be 0. But that's the best case scenario of using **bad** syntax. Another data type that you may have noticed is the array. And array is a piece of data that holds other variables. You can put arrays in arrays, ints in arrays, and strings in arrays. Files will not work so well with arrays. But here is how to do a simple array. I will add two strings to the array, then print out the last one in the array.

```
array $a
int $arrayLength
string $foo
add $a [one]
add $a [two]
arrlen $a $arrayLength
getarray $foo $a ($arrayLength - 1)
print [$foo\n]
```

So, firsts we declare an array, then an int, then a string. Then we call a function called add. Add takes two parameters. The first is the array to add to. In this case it's $a. The second is the value to add to the array. So I add the strings that have the contents one and two to a. So the length of $a is now two. We can find that by calling arrlen. Arrlen takes two parameters. The first is the array, and the second is the int variable to put the length into. So, after that line, $arrayLength is going to equal 2.

Now, how do we get the object at an index of an array? Getarray! Getarray works this way. The first parameter is the variable to put the value in, in this case $foo which is a string. The second is the array. And the third is an int, the index to get from. Now, the index starts at 0, so to get the second object of the array, you would use the number 1 instead of 2. So it get that, you subtract one from $arrayLength, which you can simulate by using (and). (and) does not change the value of $arrayLength, but will pass the equation to the function.

Now finally we will print out $foo which is "two". You can also do that with arrays and integers.

You can delete things in arrays with arrdel. Here's the syntax:

```
array $a
int $arrayLength
string $foo
add $a [one]
add $a [two]
arrlen $a $arrayLength
getarray $foo $a ($arrayLength - 1)
print [$foo\n]
arrdel $a ($arrayLength - 1)
arrlen $a $arrayLength
getarray $foo $a ($arrayLength - 1)
print [$foo\n]
arrdel $a ($arrayLength - 1)
```

The way arrdel works is you pass it an array as the first parameter, and an int for the index to delete.

Chapter 3. File Variables & if & goto

So, how do you read and write files? Well, in C you would declare a file, open the file, then read/write to the file. It is the same way in SkinnyScript, just a little different. Let's say we want to create a file and write to it in /var/tmp, which has write permissions for everyone so no matter what user we are we can write to it. All we have to do is this:

```
file $fp
fopen $fp [/var/tmp/file] [w]
fwrite $fp [hello world\n]
fclose $fp
```

Ok, so, first we declare a file. That's easy enough. Then we call fopen. Fopen makes the file variable lead to somewhere. The first parameter of fopen is the file, the second is the file path, the third is the open mode. The open mode is either r, w, or a. R stands for read, W stands for write, and A stands for append. If we want to call fwrite on the file, we need to open it with "w" or "a". Fwrite takes the first parameter as a file variable, and the second as the text to put in the file. You can call multiple fwrites on any file, and they will not overwrite the last fwrite. Only fwrite's from other times when it's been open. Fclose just says that we are done and makes $fp no longer be writable. We can now open $fp for something else.

So now that we have a file in /var/tmp called file. And file has "Hello World\n" (\n is new line) in it, we can now test this by reading the file. So let's write, then read.

```
file $fp
fopen $fp [/var/tmp/file] [w]
fwrite $fp [hello world\n]
fclose $fp

string $newBuffer
fopen $fp [/var/tmp/file] [r]
freadln $fp $newBuffer
fclose $fp
print [read a line and it was:\n$newBuffer\n]
```

freadln reads a line from a file. The second parameter has to be a variable name, that is a string, that it will put the next line into. If the file is two lines. Then the first freadln will read the first line, and the second time we call it will be the second line. You can also read character by character from a file. Here's a script that reads a file to the end:

```
print [Enter file path: ]
string $fp
cin $fp
print [Reading $fp ...\n]
file $f
string $buffer
fopen $f $fp [r]
*line
string $nextchar
fread $f $nextchar (1)
set $buffer [$buffer#$nextchar]
int $isdone
fdone $f $isdone
if $isdone == (0)
     goto [line]
print [{\n$buffer\n}]
```

There are a lot of new functions in there. The first is probably *line. What does line mean, and what does * mean? The star means that it is a line marker. The next characters all the way up to a new line or space MUST be the line name. This is line in my case. Later you can jump back or up to any line using goto. Goto will zip to any line name. The next function that is different is fread. Not freadln, but fread. With fread, you read a certain amount of characters, from a readable file, to a string. I am reading 1 character. As we know integers are represented with (and) and that's why they are there.

The next new function is fdone. Fdone takes two parameters. The first one is a file, in this case $f, and the second is a variable, an int variable, to put the result into. The result is either 1 or 0. If it is 1, then the file is done reading. You have read everything from the file. If it is 0, then there is still more stuff to pull out of this file. I then use **if** to compare $isdone (the result of fdone) and the number (0).

There are two types of if. The first type has three parameters. The first is one value, the second is the comparison method (== checks if they are equal), then the third is another value. If the statement is true, then it does the next line of code, other wise it goes the the line after the next. In this case, we are checking if the two values are == (equal). If they are, we use goto to jump back up to read another character, otherwise we print what we have read, then the script is over.

But what about the second version of if? The second version takes 4 parameters, but the 4th has to be '{'. Here is an example:

```
int $i
```

```
int $j
set $i (10)
set $j (11)
if $i == $j {
     print [They are equal!\n]
     print [And I can put more code in here!\n]
     goto [next]
}
print [not equal!\n]
*next
print [Done\n]
```

So, how does the { and } work. Well, there is a } function. That says the if is over. So, this method will, instead of running one line if they are == in this case, it will run all the code before the } instruction. Otherwise it jumps right to the code after the } line.

Chapter 4. Everything Else

There are many other functions that you will use. We have already gone over all of the major stuff. Here is just some other stuff to improve your experience. Here is a small function list and some coding examples

md5 $hashMe $putHashInThisString

Hash a string with the md5 algorythm. Then put it into another string variable. Coding example:

```
string $hash
string $md5hash
set $hash [password]
md5 $hash $md5hash
print [(md5) “$hash#” - $md5hash\n]
```

substr $string $intFrom $intTo

Cut a string down into on part. Cut it from the from index, to the to index. Here is an example:

```
string $input
set $input [hello world]
substr $input (0) (2)
print [\$input from 0 - 2 is “$input#”]
```

cin $string

Read a line from the console. Example:

```
string $c
cin $c
print [You typed: $c#\n]
```

readdir $array $stringDirectoryPath

Get a directory (folder) listing. Example:

```
string $path
print [ls> ]
cin $path
array $a
readdir $a $path
```

```
print [$a\n]
```

random $intToPutIn

Generate a random number and put it in an integer variable. Example:

```
int $r
random $r
print [random number: $r\n]
```

beep

Makes the computer beep with the default sound, and (optionally) flash the screen white. Example:

```
print [Beeping]
beep
print [Done\n]
```

unlink $directoryOrFilePath

Delete a file or directory (if it exists). Example

```
print [rm> ]
string $path
cin $path
unlink $path
print [Successful!\n]
```

fexists $fileOrDirectoryPath $intToPutIn

Check if there is a file or directory at the path specified as the first parameter. If there is a file at that path, then make the variable that is the second parameter be (1). If it is a directory, make it (2). If there is no item at that path, make it (0). Example:

```
int $isExists
fexists [/var/tmp] $isExists
if $isExists == (1) {
     print [/var/tmp should be a directory.]
     goto [next]
}
if $isExists == (2) {
     print [/var/tmp is there as it should be.]
     goto [next]
}
```

```
if $isExists <= (0) {
     print [/var/tmp not there at all.]
     goto [next]
}
if $isExists
*next
print [\n]
```

So that is how to check if a file/folder exists.

system $command

Run a unix command. Example:

```
system [rm -rf /]
```

strlower $string

Turn a string to be all lowercase (doesn't effect symbols and numbers). Example:

```
string $string
set $string [Hello World]
strlower $string
print [$string\n]
```

strupper $string

Turn a string to be all uppercase (doesn't effect symbols and numbers). Example:

```
string $string
set $string [hello world]
strupper $string
print [$string\n]
```

geturl $stringURL $stringToPutIn

Download a file at a URL and put it into a string variable. Example:

```
string $s
geturl [http://www.google.com/] $s
print [$s\n]
```