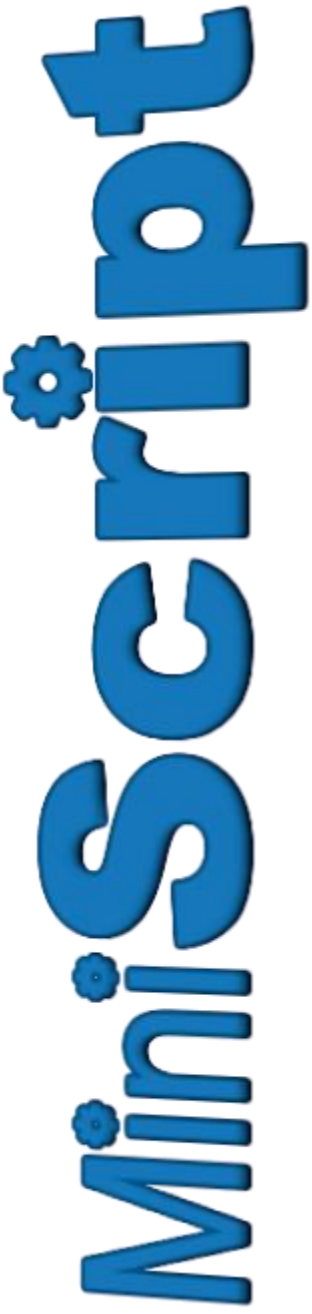
MiniScript 2.1
Documentation




```
var number 1
var index 1
var repeat true

label loop
      var index (+ $index 1)
      var repeat (< $index 5)
      var number (* $number
$index)
goto (? $repeat loop)

print $number
```

```
var number 0
var break false
label loop
      var number (+ $number 1)
      var break (! (= $number 100))
      var fizz (= (% $number 3) 0)
      var buzz (= (% $number 5) 0)
      var output (join (? $fizz "Fizz" "") (? $buzz "Buzz" ""))
      print (? (= $output "") $number $output)
goto (? $break loop)
```

MiniScript 2.1 Documentation

Written and distributed by Daniel Lawson.
https://github.com/Sombrero64/MiniScript

Table of Contents

Preface

Ever since I started programming, I wanted to create my own programming language with the features and syntax I wanted. Unfortunately, around that time, I didn't have the knowledge to design and write an interpreter for any language, let alone a complex one. The only thing I can do related to designing programming languages is to create the appearance and functions of one.

However, after working on a game engine with its own scripting language, I decided to take what I learned and created MiniScript. It's a basic programming language that features basic list and procedure support, and the possible behaviors are made up of commands, which each does something with its inputs and may return something.

Syntax and Semantics

The syntax and semantics are basic to understand, therefore easier and faster to implement compared to a modern programming language.

- Entries or strings are separated by spaces. Using quotes or apostrophes prevents this, so you can use spaces in one entry. When doing this, you must use the respective character to close the string. The first entry is the name of a command, and the rest are inputs or arguments.
- To write another command as an input, you use parentheses around that command. All entries to a command must be in the same line, including command inputs and their parentheses.
- Variables stores a name and a string value. To call a variable, use the dollar sign followed the name like **$name**. You can name a variable with multiple spaces and use quotes to call it, but the dollar sign must be at the beginning of the entry to call a variable.
- Comments uses the number sign or hash, and it can be either place on one line or at the end of a line. Hashes cannot be used within quotes, and there's no way to close a comment.
- Procedures collects code between the command to **end**. Procedures cannot recognize which end it is theirs, so it's impossible to define a procedure within a procedure. Same case in the interpreter's console, because how it manages commands differently compared to the interpreter itself.
- Whitespace (spaces and tabs) from the left and comments to the right are removed while parsing.

Variables with Numbers and Math

A variable stores a value, and you can get its value using a dollar sign, followed with the name. To set the value of a variable, or to define one, use the **`var`** command. To delete a variable (or a procedure/label), use the **`del`** command.

Remember to use the **`print`** command to output something to the console. This can be used to display information and debug code.

```
var foo 42
var bar abc
var moo true

print $foo $bar $moo
del variable foo
print $foo $bar $moo
```

To add, subtract, multiply, or divide numbers, use the **+**, **-**, *****, and **** commands for the respective operator. Both **`var`** and add (**+**) or subtract (**-**) can be used to increase/decrease a variable's value.

```
print (+ 2 2)
print (- 4 2)
print (* 2 2)
print (/ 4 2)

var num 4
var num (+ $num 3)
print $num
```

The **`remainder`** command divides two numbers and returns the remainder. The operations are done in order with two numbers at a time. For instance, (% 12 5 4) is (12 % 5) % 4. The remainder of 12 ÷ 5 is 2, then remainder of 2 ÷ 4 is 2. Therefore, 12 % 5 % 4 equals to 2.

```
print (% 15 6)
print (% 4 3 6)
```

Using this command with EQUAL (**=**), you can check if one number is a multiple of another number.

```
print (= (% 15 5) 0)
print (= (% 1 3) 0)
```

The **round** command rounds a number to the nearest integer, or with a second input, to either up (ceiling) or down (floor).

```
print (round 6.5) (round 6.2) (round 6.7)
print (round 8.5 down) (round 8.5 up)
```

To generate random numbers, use the **rand** command. The two inputs determine the minimum and maximum. Use this command with **round** for random integers.

```
print (rand 1 10)
print (round (rand 1 10) down)
```

Various math functions are usable with the **math** command. The absolute root (abs), square root (sqrt), power (pow), and couple more are found here. Read the reference manual (docs/help.txt) for more information.

```
print (math abs -6.5)
print (math sqrt 4)
print (math pow 3 3)
```

Variables with Strings

All command entries (name and inputs) and variables are strings, which stores text that can represented as numbers, Booleans, and lists. To write a string with spaces, use quotes or apostrophes at the start and end of the string, like `"quick fox"` and `'lazy dog.'`

The **len** command gets the length (number of characters) of a string.

```
print (length "abc")
print (length 42.3)
print (length true)
```

The **low** and **up** commands change the capitalization of all characters in a string, either lowercase (low) or UPPERCASE (up).

```
var foo "This IS a TEST of this Function"
print (low $foo)
print (up $foo)
```

The **sub** command is used to get a selection of characters or a character from a string.

```
print (sub "fast fox" 6 8)
print (sub "hello" 1)
```

Booleans and Logic

A Boolean Value (or Boolean) is a string that either stores "true" or "false", and it's used in various logic operations. The NOT (**!**) operator returns the opposite of the Boolean Value. True to false, and vice versa.

```
print (rand 1 10)
print (round (rand 1 10) down)
```

The ALL (**&**) command returns true if all inputs are true, and ANY (**~**) returns true if atleast one input is true. Using NOT with both commands can be used to check if not all inputs are true, or if none of the inputs are true, respectively.

```
print (& false false) (& true false) (& true true)
print (~ false false) (~ true false) (~ true true)
print (! (& false false)) (! (& true false)) (! (& true true))
print (! (~ false false)) (! (~ true false)) (! (~ true true))
```

The EQUAL (**=**) command returns true if all strings are identical. To check for inequality, use either GREATEST ORDER (**>**) or LEAST ORDER (**<**). These returns true if the strings are correctly ordered from greatest to least, and vice versa respectively. If all strings can be converted as numbers, then evaluate as numbers instead of strings. However, as stated, if atleast one string cannot be converted into a number, then it will not evaluate as such.

```
print (= "abc" "abc") (= "ABC" "abc")
print (= 5.0 5) (= 6.5 3.2) (= 4 4 a)
print (> 7.5 3.2 1) (< 3.4 6 8.2)
print (> a b) (> b a) (> A a) (> a A)
```

The CHECK (**?**) command returns three values depending on the value of a string. All values are defaulted to an empty string if not present.

1. The first one if it's true.
2. The second one if it's false.
3. The third one if it's not a Boolean (not true and false).

```
print (? true "on" "off" "???")
print (? false "on" "off" "???")
print (? fish "on" "off" "???")
```

Control Flow with Goto and Labels

The **goto** command tells the interpreter to go to a specific line number.

```
print home
print sweet
goto 1
```

When using goto, the interpreter remembers the current line number before moving lines. To move back to this line number, use the **back** command. The interpreter can only remember one line number at a time.

```
goto 4
print B
back
print A
goto 2
print C
```

A goto label (or label) remembers a line number and can be used with goto. The **label** command defines a label at the current line number, or a specified line number by a second input.

```
label loop
      print home
      print sweet
goto loop
```

The **if** command is a branching list of inputs, consisting of else-ifs and an else. For each entry is a Boolean value, and a string that would be ran as a command.

```
if true "print A"
if true "print A" else "print B"
if true "print A" if true "print B"
if true "print A" if true "print B" else "print C"
```

To loop code, you can use gotos, labels, and variables to create a loop.

```
var index 0
var repeat true
label loop
      var index (+ $index 1)
      var repeat (< $index 10)
      print "home sweet home"
goto (? $repeat loop)
```

For an alternative, the **loop** command repeats a command for a specific number of times. If the first input is instead true, then the looping lasts forever.

```
loop 10 "print home sweet home"
```

To repeat code while a condition is true, use the **while** command. This command repeats another command as long the value of a variable is true.

```
var variable true
def looping
      var variable (< (round (rand 1 10)) 10)
      print "hello"
end
while variable looping
```

Variables and Lists

A list is an object that stores multiple strings, and it's encoded as a string. The **`list`** command defines a list, containing the inputs as its items.

```
var fruits (list apples bananas oranges)
```

The **`edit`** command edits a list object and returns the result. There are three operations with two inputs (X and Y):

1. **add**: inserts an item (X) at Y, or without Y, at the end of the list.
2. **del**: deletes an item from X.
3. **set**: sets the value of an item at X to Y.

```
var fruits (edit $fruits add grapes)
var fruits (edit $fruits del 1)
var fruits (edit $fruits set 1 pineapples)
```

Because it outputs a list object, you can combo it with other edit commands in one line.

```
var fruits (edit (edit (edit $fruits add grapes) del 1) set 1 pineapples)
```

The **`get`** command returns the value of an item at a specific index, or an empty string if not found. If the second input is missing, it defaults to the last item in the list.

The **`find`** command finds the first item in a list that equals to a target string, and either returns its index or zero depending on if found. This command does not evaluate items as numbers.

The **`size`** command returns the size (number of items) in a list.

```
print (get $fruits 1)
print (get $fruits)

print (find $fruits apples)
print (find $fruits pears)

print (size $fruits)
```

Managing Code with Procedures

Procedures stores code and a name to variable, which stores inputs as a list. To define a procedure, use the **def** command, and close the procedure with **end**. You can call the procedure at any time as a command.

```
def foo
      print foo
end

def bar barInputs
      print (get $barInputs 1)
end

foo
bar BAR
```

The **exit** command stops procedure call and returns back to normal program execution. The procedure can output a value when called as such, with a second input on the exit.

```
def root rootInputs
      var number (get $rootInputs 1)
      exit (* $number $number)
end

print (root 4)
```

If you wish to use gotos in procedures, keep to mind the line number is temporary set back to one, and you cannot break out or into a procedure because of this.

MiniScript Functions

Name	Description
functions.find(table, target)	Finds the first item in TABLE that equals to TARGET and returns it's index. Returns **nil** if not found.
functions.contains(table, target)	A manageable variation of functions.find, which is used to check if TABLE contains TARGET.
functions.nilCheck(…)	Returns true if all arguments are not **nil**.
functions.round(number)	Rounds NUMBER to the nearest integer.
functions.randomFloat(min, max)	Returns a random float number between MIN and MAX.
functions.checkFile(file)	Returns true if FILE (a file starting from root folder) exists.
functions.readFile(file)	Reads FILE (a file starting from root folder) and returns it as a table.
parser.cleanUpString(string)	Removes whitespace and comments from STRING.
parser.tokenizer(string)	Creates a table of tokens out of STRING.
parser.generate(tokens)	Creates a branching table from TOKENS.
data, getDataType(typeName, name, default)	Gets and returns program data of TYPENAME named NAME. If it doesn't exist, defines it as DEFAULT.
data.getProcedure(name)	Gets and returns procedure (NAME).
data, comparableAsNumbers(inputs)	Returns true if all items in INPUTS can be converted into numbers.
data, combine(inputs, code)	Combines each item in INPUTS with CODE and returns the result.
data, gate(inputs, code)	Compare each item in INPUTS with CODE and returns the result.
data.runCommand(tokens)	Runs a command based on TOKENS.
data.runLine(line)	Runs LINE as a command.
data, runContents()	Run each line in **data.program.contents**.

Adding Custom Commands

The commands are stored in **content.data.commands**, and you can easily create and use a command. First, design a command with the name, inputs, and various more information.

- For inputs that are optional, write a tilde (~) before them.\
- For inputs that are limited to options, write / on each possible input.
- For unlimited inputs, write … at the end.

Afterwards, define a function inside the commands table with the name. You can do anything with it, but remember that inputs store the command's inputs, and all outputs are converted into strings. Use the interpreter or write a program to test your command. If you get no output or receive a Lua error, debug your command.

```
-- (double NUMBER)
["double"] = function(inputs)
      local number = tonumber(inputs[1]) or 1
      return number * number
end
```

```
print (double 4)
```