Welcome to MiniScript!

MiniScript is a high-level object-oriented language that is easy to read and write.

Clean Syntax

Put one statement per line, with no semicolons, except to join multiple statements on one line.

Code blocks are delimited by keywords (see below). Indentation doesn't matter (except for readability).

*Comments begin with **//**.*

*Don't use empty parentheses on function calls, or around conditions in **if** or **while** blocks.*

All variables are local by default. MiniScript is case-sensitive.

Control Flow

if, else if, else, end if

*Use **if** blocks to do different things depending on some condition. Include zero or more **else if** blocks and one optional **else** block.*

```
if 2+2 == 4 then
   print "math works!"
else if pi > 3 then
   print "pi is tasty"
else if "a" < "b" then
   print "I can sort"
else
   print "last chance"
end if
```

while, end while

*Use a **while** block to loop as long as a condition is true.*

```
s = "Spam"
while s.len < 50
   s = s + ", spam"
end while
print s + " and spam!"
```

for, end for

*A **for** loop can loop over any list, including ones easily created with the **range** function.*

```
for i in range(10, 1)
   print i + "..."
end for
print "Liftoff!"
```

break & continue

*The **break** statement jumps out of a **while** or **for** loop. The **continue** statement jumps to the top of the loop, skipping the rest of the current iteration.*

Data Types

Numbers

All numbers are stored in full-precision format. Numbers also represent true (1) and false (0). Operators:

+, -, *, /	standard math
%	mod (remainder)
^	power
and, or, not	logical operators
==, !=, >, >=, <, <=	comparison

Strings

Text is stored in strings of Unicode characters. Write strings by surrounding them with quotes. If you need to include a quotation mark in the string, type it twice.

```
print "OK, ""Bob""."
```

Operators:

+	string concatenation
-	string subtraction (chop)
*, /	replication, division
==, !=, >, >=, <, <=	comparison
[i]	get character i
[i:j]	get slice from i up to j

Lists

*Write a list in square brackets. Iterate over the list with **for**, or pull out individual items with a 0-based index in square brackets. A negative index counts from the end. Get a slice (subset) of a list with two indices, separated by a colon.*

```
x = [2, 4, 6, 8]
x[0]   // 2
x[-1]  // 8
x[1:3] // [4, 6]
x[2]=5 // x now [2,4,5,8]
```

Operators:

+	list concatenation
*, /	replication, division
[i]	get/set element i
[i:j]	get slice from i up to j

Maps

A map is a set of values associated with unique keys. Create a map with curly braces; get or set a single value with square brackets. Keys and values may be any type.

```
m = {1:"one", 2:"two"}
m[1]   // "one"
m[2] = "dos"
```

Operators:

+	map concatenation
[k]	get/set value with key k
.ident	get/set value by identifier

Functions

*Create a function with **function()**, including parameters with optional default values. Assign the result to a variable. Invoke by using that variable. Use @ to reference a function without invoking.*

```
triple = function(n=1)
   return n*3
end function
print triple      // 3
print triple(5)   // 15
f = @triple
print f(5)   // also 15
```

Classes & Objects

*A class or object is a map with a special **__isa** entry that points to the parent. This is set automatically when you use the **new** operator.*

```
Shape = {"sides":0}
Square = new Shape
Square.sides = 4
x = new Square
x.sides  // 4
```

*Functions invoked via dot syntax get a **self** variable that refers to the object they were invoked on.*

```
Shape.degrees = function()
   return 180*(self.sides-2)
end function
x.degrees   // 360
```

Intrinsic Functions

Numeric

```
abs(x)     acos(x)  asin(x)
atan(y,x)  ceil(x)  char(i)
cos(r)     floor(x) log(x,b)
round(x,d) rnd      rnd(seed)
pi         sign(x)  sin(r)
sqrt(x)    str(x)   tan(r)
```

String

```
.indexOf(s)     .insert(i,s)
.len        .val     .code
.remove(s) .lower   .upper
.replace(a,b)   .split(d)
```

List/Map

```
.hasIndex(i)    .indexOf(x)
.insert(i,v)    .join(s)
.push(x)    .pop     .pull
.indexes    .values
.len        .sum     .sort
.shuffle    .remove(i)
range(from,to,step)
```

Other

```
print(s)   time     wait(sec)
locals     outer    globals
yield
```