MiniScript Manual

Learn to read, write, and speak the world's easiest computer language

Joe Strout

MiniScript Master

Version 1.3

Friday, March 1, 2019

Welcome to MiniScript

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

MiniScript is a modern scripting language designed to be clean, simple, and easy to learn. It was designed from the ground up by borrowing only the best ideas from other languages such as Python, Lua, Basic, and C#. If you know pretty much any other programming language, you'll pick up MiniScript almost immediately.

And if you've never written a line of code in your life, don't panic! MiniScript is the friendliest and most fun way to get started. It's much easier than you probably expect.

Important: MiniScript is designed as an *embedded* programming language. That means you will usually use it inside some other program, such as a video game. You should find another document that describes how to access and use MiniScript within that other program. This document only describes the MiniScript language itself, and the intrinsic functions that are common to most MiniScript applications.

Clean, Clear Syntax

Let's jump right in with an example, to see what MiniScript code looks like.

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

Each statement in MiniScript normally occurs on a single line by itself. Notice that there are no semicolons, curly braces, or other markers at the end of a line.

However, there is one exception: if you want to join multiple statements on one line, just to make your code more compact, you can do this by separating the statements with a semicolon. The following code is ugly, but legal.

```
s = "Spam"; while s.len < 50; s = s + ", spam"; end while

print(s + " and spam!")
```

In practice this feature is rarely used, but it's there if you need it.

Code Blocks

If you're used to C-derived languages (such as C, C++, C#, etc.), then you're used to seeing curly braces around blocks of code. MiniScript doesn't roll that way; code blocks always begin with a keyword (`if`, `for`, `while`, or `function`) and end with a matching end statement (`end if`, `end for`, `end while`, or `end function`).

Whitespace and Indentation

You can insert spaces and tabs into your code pretty much wherever you want. You can't break up an identifier or keyword (`pr int` is not the same as `print`), nor omit a space between two identifiers or keywords (`end if` is correct, but `endif` would not work). And of course spaces within quotation marks go into your string exactly as you would expect. But between numbers, operators, etc., you can include extra spaces however you like. The following two lines are exactly the same, as far as MiniScript is concerned.

```
x=4*10+2
x = 4 * 10 + 2
```

To make the structure of code more readable, it's traditional to indent the lines within a code block by either a tab or two spaces. But it's not required. MiniScript doesn't care how or whether you indent your code, so do whatever works best for you.

Comments

Comments are little notes you leave for yourself, or other humans reading your code. They are completely ignored by MiniScript. Comments begin with two slashes, and extend to the end of a line. So you can put a comment either on a line by itself, or after a statement.

```
// How many roads must a man walk down?
x = 6*7  // forty-two
```

Just like indentation, comments are never required… but they're probably a good idea!

Use of Parentheses

Parentheses in MiniScript have only three uses:

1. Use them to group math operations in the order you want them, just as in algebra.

```
x = (2+4)*7   // this is different from 2+4*7
```

2. Use them around the arguments in a function call.

```
print(42)    // parens required here; print 42 would not work
```

3. Use them when declaring a function (see the Functions chapter.)

Since other languages often require parentheses in lots of other silly places, it's worth pointing out where parentheses are *not* used in MiniScript. First, don't put parentheses around the condition of an `if` or `while` statement (more on these later). Second, parentheses are not needed (and should be omitted) when calling a function without any arguments. For example, there is a `time` function that gets the number of seconds since the program began. It doesn't need any arguments, so you can invoke it without parentheses.

```
x = time
```

This also applies to functions like `print` and `wait`, whose arguments are optional. Use parentheses if passing in a value; otherwise, leave them off. The following example prints ten blank lines, waiting one second each, and then prints a message. Notice how we're calling `print` and `wait` without any parentheses.

```
for i in range(1,10)
  print
  wait
end for
print("Boom!")
```

Local and Global Variables

A *variable* is a word (also called an identifier) associated with a value. Think of variables as little boxes that you can store data in. You create a variable simply by assigning a value to it, as in many of the examples we've already seen.

```
x = 42
```

This line creates a variable called `x`, if it didn't exist already, and stores 42 in it. This replaces the previous value of `x`, if any.

Variables in MiniScript are **dynamically typed**; that is, you can assign any type of data (see the chapter on Data Types) to any variable.

Variables are always **local** in scope. That means that a variable called "x" inside one function has nothing at all to do with another variable called "x" in another function; each variable is scoped (restricted) to the current function executing at the time of the assignment.

However, MiniScript also supports code outside of any function, as in all the examples we've seen so far. In this context, local and global variables are the same. In other words, assigning 42 to `x` outside of a function creates a global variable called `x`. Such global variables may be accessed from any context.

Note that when a context has a local variable of the same name as a global, an identifier will always resolve to the local variable first. Similarly, a simple assignment statement within a function will always create a local variable, rather than a global one. In cases where you really need to access the global variable instead, there is a `globals` object that provides this access. (See the Intrinsic Functions chapter for more detail on `globals`.)

```
demo = function()
  print(x)          // prints the global x (40)
  x = 2             // creates a local ‘x’ with a value of 2
  print(x)          // prints the local x (2)
  print(globals.x)  // prints the global x again (40)
  globals.x = 42    // reassigns the global x
  print(x)          // still the local value (2)
  print(globals.x)  // prints the new global value (42)
end function

x = 40   // creates a global ‘x’ with a value of 40
demo     // invokes the function above
```

Overuse of global variables can sometimes lead to tricky bugs, so it’s best to use them sparingly and rely on local variables as much as possible. MiniScript is designed so that this good practice is what happens naturally.

MiniScript is Case-Sensitive

Uppercase and lowercase matters in MiniScript. The `print` intrinsic function must be typed exactly `print`, and not `Print`, `PRINT`, or any other variation. The same applies to any variables, functions, or classes you define.

While how you use case in your own identifiers is up to you, a common convention is to capitalize classes (e.g. Shape), but use lowercase for variables. Thus the following would be a perfectly sensible bit of code.

```
shape = new Shape   // create a Shape object called shape
```

While we're on the subject of conventions, in most cases you should avoid starting any global variables or function names with an underscore. Identifiers starting with an underscore are often used by the host environment for special “under the hood” code, and name collisions there could cause problems.

Control Flow

looping and branching

Control flow is how you make code execute multiple times, or execute only under certain conditions. Without it, your scripts would be limited to starting at the first line, executing each line exactly once in order, and ending after the last line.

MiniScript includes one kind of branching (conditional) structure, and two kinds of loops.

Branching with if

Use an `if…then` statement to specify some condition under which the following statements should be executed. The basic syntax is:

> `if` *condition* `then`
> …
> `end if`

When the condition is not true, MiniScript will jump directly to the `end if` statement.

```
if x == 42 then
  print("I have found the Ultimate Answer!")
end if
```

The whole set of lines, from `if…then` to `end if`, is known as an *if block*.

Sometimes you want to do something else when the specified condition is not true. You can specify this with an *else block* before the `end if`.

```
if x == 42 then
  print("I have found the Ultimate Answer!")
else
  print("I am still searching.")
end if
```

Finally, you can check for additional conditions by adding *else-if blocks* as needed. Here's a slightly more practical example that converts a number to words.

```
if apples == 0 then
  print("You have no apples.")
else if apples == 1 then
  print("You have one apple.")
else if apples > 10 then
  print("You have a lot of apples!")
else
  print("You have " + apples + " apples.")
end if
```

In this case, the first condition that matches will execute its block of lines. If none of the conditions match, then the `else` block will run instead.

Note that for all these forms, the `if`, `else if`, `else`, and `end if` statements must each be on its own line. However, there is also a "short form" `if` statement that allows you to write an `if` or `if/else` on a single line, provided you have only a single statement for the `then` block, and a single statement for the `else` block (if you have an `else` block at all). A short-form `if` looks like this:

```
if x == null then x = 1
```

...while a short-form `if/else` looks like this:

```
if x >= 0 then print("positive") else print("negative")
```

Notice that `end if` is not used with a short-form `if` or `if/else`. Moreover, there is no way to put more than one statement into the `then` or `else` block. If you need more than one statement, then use the standard multi-line form.

Looping with for

A `for…in` statement loops over a block of code zero or more times. The syntax is:

> `for` *variable* `in` *list*
>
> …
>
> `end for`

The whole block is referred to as a *for loop*. On each iteration through the loop, the variable is assigned one value from the specified list. You'll learn more about lists in the Data Types chapter, but for now, it's enough to know that you can easily create a list of numbers using the `range` function.

This example counts from 10 down to 1, and then blasts off.

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

See the `range` function in the Intrinsic Functions chapter for more options on that.

Instead of a list, you can also iterate over a text string. In this case the loop variable will be assigned each character of the string in order.

Finally, it is also possible to iterate over maps. Again, maps will be explained in the Data Types chapter, but just keep in mind that when you use a `for` statement with a map, then on each iteration through the loop, your loop variable is a little mini-map containing `key` and `value`. For example:

```
m = {1:"one", 2:"two", 3:"three"}
for kv in m
  print("Key " + kv.key + " has value " + kv.value)
end for
```

This prints out each of the key-value pairs in the map.

Looping with while

The other way to loop over code in MiniScript is with a *while loop*. The syntax is:

> `while` *condition*
>   …
> `end while`

This executes the contained code as long as *condition* is true. More specifically, it first evaluates the condition, and if it's not true, it jumps directly to `end while`. If it is true, then it executes the lines within the loop, and then jumps back up to the `while` statement. The process repeats forever, or until the condition becomes false.

This is illustrated by the very first example in this manual, repeated here.

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

This code builds a string (s) by adding more spam to it, as long as the string length is less than 50. Once it is no longer less than 50, the loop exits, and the result is printed.

Break and Continue

There are two additional keywords that let you bail out of a while or for loop early. First, the `break` statement jumps directly out of the loop, to the next line past the `end for` or `end while`. Consider the following.

```
while true     // loops forever!
  if time > 100 then break
end while
```

Whenever you see `while true` (or `while 1`, which is equivalent), it is an infinite loop — *unless* there is a `break` statement in the body of the loop. As soon as that `break` statement executes, we jump directly out of the loop. It works for `for` loops in exactly the same way. In the case of nested loops, `break` breaks out of only the innermost loop.

The `continue` statement skips the rest of the body of the loop, and proceeds with the next iteration. This is often used for "bail-out" cases in a large loop, where under certain conditions you want to skip an iteration and just go on with the next one.

```
for i in range(1,100)
  if i == 42 then continue
  print("Considering " + i + "...")
end for
```

This will print out the numbers 1 through 100, *except* for 42, which is skipped. Note that if you simply changed `continue` to `break` in this example, the loop would print the numbers 1 through 41, and then stop.

The Nature of Truth

We have talked about evaluating conditions as true or false, without explaining what that really means. Usually you don't need to worry about it, but here are the details anyway.

Boolean (true/false) values in MiniScript are represented as numbers. When conditions are evaluated for `if` and `while` statements, a value of 0 (zero) is considered false; and other value is considered true. In fact the built-in keywords `true` and `false` are exactly equivalent to the numbers 1 and 0 respectively.

When you use comparison operators such as `==` (equal), `!=` (not equal), `>` (greater than), and `<=` (less than or equal), these compare their operands and evaluate to either 1 (if true) or 0 (if false).

See the Numbers section of the Data Types chapter for more boolean operations you can apply to numbers (including `and`, `or`, and `not`).

Finally, in a context that demands a truth value — that is, in an `if` and `while` statement, or as an operand of `and`, `or`, and `not` — other data types will be considered false if they are empty, and true if they are not empty. So an empty string, list, or map is equivalent to 0 (zero), and any non-empty string, list or map is equivalent to 1 in these contexts. The special value `null` is always considered false.

Data Types

things you can store and manipulate

Variables in MiniScript are dynamically typed; you can store any type of data in any variable. But what types of data are there? In MiniScript, there are four primary data types: *numbers*, *strings*, *lists*, and *maps*. There are a couple of other more obscure types, such as function and null. Everything else, including classes and objects, is actually a special case of a map.

Numbers

All numeric values in MiniScript are stored in standard full-precision format (also known as "doubles" in C-derived languages). Numbers are also used to represent true (1) and false (0).

Numeric literals are written as ordinary numbers, e.g. `42`, `3.1415`, `-0.24`.

You can use the following operators on numbers (where *a* and *b* are numbers).

a + b	addition	numeric sum of a and b
a - b	subtraction	numeric difference of a and b
a * b	multiplication	a multiplied by b
a / b	division	a divided by b
a % b	modulo	remainder after dividing a by b
a ^ b	power	a raised to the power of b
a and b	logical and	a * b, clamped to the range [0,1]
a or b	logical or	a + b - a*b, clamped to the range [0,1]
not a	negation	1 - abs(a), clamped to the range [0,1]
a == b	equality	1 if a equals b, else 0
a != b	inequality	1 if a is not equal to b, else 0
a > b	greater than	1 if a is greater than b, else 0
a >= b	greater than or equal	1 if a is greater than or equal to b, else 0
a < b	less than	1 if a is less than b, else 0
a <= b	less than or equal	1 if a is less than or equal to b, else 0

Note that `and`, `or`, and `not` are not functions; they are operators, and go between (or in the case of `not`, before) their operands just like all the others.

Strings

Text values in MiniScript are stored as strings of Unicode characters. String literals in the code are enclosed by double quotes ("). Be sure to use ordinary straight quotes, not the fancy curly quotes some word processors insist on making.

If your string literal needs to include quotation marks, you can do this by typing the quotation marks twice. For example:

```
s = "If you do not help us, we shall say ""Ni"" to you."
```

Strings may be concatenated with the + operator, and if you try to add a number and a string together, the number will be automatically converted to a string and then concatenated. Strings may also be replicated (repeated) or cut down to a fraction of their former selves, by multiplying or dividing them by a number.

```
s = "Spam" * 5   // SpamSpamSpamSpamSpam
s = s / 2        // SpamSpamSp
```

The full set of string operators is shown below; *s* and *t* are strings, and *n* and *m* are numbers.

s + t	concatenation	string formed by concatenating t to s
s * n	replication	s repeated n times (including some fractional amount of s)
s / n	division	equivalent to s * (1/n)
s[n]	index	character n of s *(all indexes are 0-based; negative indexes count from end)*
s[:n]	left slice	substring of s up to but not including character n
s[n:]	right slice	substring of s from character n to the end
s[n:m]	slice	substring of s from character n up to but not including character m
s == t	equality	1 if s equals t, else 0 *(all string comparisons are case-sensitive)*
s != t	inequality	1 if s is not equal to t, else 0
s > t	greater than	1 if s is greater than (collates after) t, else 0
s >= t	greater than or equal	1 if s is greater than or equal to t, else 0
s < t	less than	1 if s is less than (collates before) t, else 0
s <= t	less than or equal	1 if s is less than or equal to t, else 0

The table above does not include `and`, `or`, and `not`, but these operators work perfectly well on strings through boolean coercion (see "The Nature of Truth" in the previous chapter). In any boolean context, s is considered true if it contains any characters, and false if it is the empty string.

The slice operators deserve a bit of explanation. The basic syntax is `s[n:m]`, which gets a substring of s starting at character n, and going up to (but not including) character m, where we number characters starting from 0. But this basic syntax is extended with a handful of neat tricks:

1. You may specify just a single index, leaving out the colon, to get a single character. Thus `s[0]` is the first character, `s[1]` is the second, etc.

2. You may use a negative index, and it will count from the end. So `s[-1]` is the last character, `s[-2]` is the next-to-last, etc. This works for any of the slice indexes.

3. You may omit the first index from the two-index form, and it will default to 0. This is a handy way to get the first n characters of a string. So `s[:3]` returns the first 3 characters of s; `s[:-3]` returns all but the last three characters of s.

4. You may omit the last index from the two-index form, and it will continue to the end of the string. Thus, `s[3:]` skips the first three characters and returns the rest of the string.

The way these indexes work results in a lot of very handy properties. For example, `s[:n] + s[n:] == s` for any value of n from 0 through `s.len`; in other words, there is a very natural syntax for splitting a string into two parts, which is a fairly common thing to do.

Finally, note that strings are **immutable**; just like numbers, you can never change a string, but you can create a *new* string and assign it to an existing variable. The following example shows one correct and one incorrect way to change “spin” into “spun”.

```
s = "spin"
s = s[:2] + "u" + s[3:]   // OK
s[3] = "u"   // no can do (Runtime Error)
```

Lists

The third basic data type in MiniScript is the *list*. This is an ordered collection of elements, accessible by index starting with zero. Each element of a list may be any type, including another list.

You define a list by using square brackets around the elements, which should be separated with commas.

```
x = [2, 4, 6, 8]
```

The code above creates a list with four elements and assigns it to x. But again, list elements don't have to be numbers; they can also be strings, lists, or maps. Here's another example.

```
x = [2, "four", [1, 2, 3], {8:"eight"}]
```

Working with a list is very much like working with a string. You can concatenate two lists with +, replicate or cut a list with * and /, and access elements or sublists using the same slice syntax. Here are the operators valid on lists, where *p* and *q* are lists, and *n* and *m* are numbers.

p + q	concatenation	list formed by concatenating q to p
p * n	replication	p repeated n times (including some fractional amount of p)
p / n	division	equivalent to p * (1/n)
p[n]	index	element n of p *(all indexes are 0-based; negative indexes count from end)*
p[:n]	left slice	sublist of p up to but not including element n
p[n:]	right slice	sublist of p from element n to the end
p[n:m]	slice	sublist of p from element n up to but not including element m

The slice operators work exactly the same way as with strings. So `p[-1]` is the last element of list p; `p[3:]` skips the first three elements and returns the rest of the list, and so forth.

However, there is one important difference: lists are **mutable**. You can change the contents of a list, by assigning to any of the slice expressions, and no matter how many different variables are referring to that list, they will all see the change. The following example illustrates.

```
a = [1, 2, 3]     // creates a list and assigns to a
b = a             // assigns that SAME list to b
a[-1] = 5         // changes the last element of our list to 5
print(b)          // prints: [1, 2, 5]
```

Because a and b both refer to the same list, any changes (*mutations*) made to that list can be seen from either variable.

If you want to be sure you have a fresh copy of a list, rather than a shared reference, a common trick is to use `[0:]` to make a slice that includes the entire list. This copies the elements into a new list. Compare the following example to the previous one.

```
a = [1, 2, 3]     // creates a list and assigns to a
b = a[0:]         // assigns a COPY of that list to b
a[-1] = 5         // changes the last element of our first list to 5
print(b)          // prints: [1, 2, 3]  (our copy hasn't changed)
```

Maps

The final basic data type in MiniScript is the *map*. A map is a set of key-value pairs, where each unique key maps to some value. In some programming environments, this same concept is called a *dictionary*.

Create a map with curly braces around a comma-separated list of key-value pairs. Specify each pair by separating the key and value with a colon, as shown here.

```
m = {1:"one", 2:"two", 3:"three"}
```

The map created here contains three key-value pairs, each mapping a number to a string (which happens to be the English word for that number in this example).

Map keys may be numbers or strings, and must be unique; if you reuse a key, the previous value is replaced. Values may be any type, including lists or dictionaries. Order within a map is not preserved; for loops iterate over a map in arbitrary order.

Maps support only a handful of operators (*d* and *e* are maps, *k* is a key, and *v* is a value):

d + e	concatenation	map formed by assigning d[k] = v for every k,v pair in e
d[k]	index	value associated with key k in d
d.k	dot index	value associated with (string) k in d

There are two ways to get and set members of a map. The first is to use the square-brackets index operator, just as with strings or lists, except that in the case of a map, the key can be a string as well as a number.

```
d = {"yes":"hai", "no":"iie", "maybe":"tabun"}
print(d["maybe"])       // prints: tabun
d["maybe"] = "kamo"
print(d["maybe"])       // prints: kamo
```

The second way is using the *dot indexer*. This works only in the special case where the key is a string that is a valid identifier: it begins with a letter, and contains only letters, numbers, and underscores. In this case you can write the key after a dot rather than enclosing it in square brackets and quotation marks — the key essentially becomes an identifier in the language. The following is functionally equivalent to the previous example.

```
d = {"yes":"hai", "no":"iie", "maybe":"tabun"}
print(d.maybe)          // prints: tabun
d["maybe"] = "kamo"
print(d.maybe)          // prints: kamo
```

This dot indexer is mostly syntactic sugar that makes accessing elements of a map easier to read and write. But there are some subtle differences in cases where the map represents a class or object, as described in the next chapter.

Functions and Classes

the building blocks of sophisticated software

A *function* is essentially a sub-program that does some particular task. We've already seen some of the functions built into MiniScript, such as `time` and `range`. There are many more of those, which will be documented in the next chapter. But the real power of a programming language comes from defining your own functions.

Beyond that, as a program grows in size and complexity, it becomes useful to start organizing it into *classes*. A class is basically a collection of functions and data, where *objects* of a class share the same functions but may have unique data.

Functions

A function in MiniScript is a special data type, at the same level as numbers, strings, lists, and maps. You can define a function with the `function` keyword, assign it to a variable, and then invoke it via that variable, just like the built-in functions. Here's an example.

```
triple = function(n=1)
  return n*3
end function
print(triple)     // prints: 3
print(triple(5))  // prints: 15
```

This declares a function that triples any value given to it, and assigns that function to a variable called `triple`. The triple function is then invoked, with and without an argument.

The syntax for declaring a function is:

> `function(`*parameters*`)`
> …
> `end function`

where *parameters* is a comma-separated list of zero or more parameters, each of the form *name* or *name=defaultValue*. When a function is invoked, arguments will be matched up to the functions by position. If fewer arguments are given than parameters are defined, the remaining parameters are given their default values — and if no default value was defined for that parameter, then it is set to `null`.

Note that the parentheses are required after the function keyword, even if no parameters are needed — one of the few cases in MiniScript where empty parentheses ever appear.

It's important to understand that a function is itself a bit of data. It's just that, whenever looking up the value of a variable, MiniScript checks for this special function data type; and if found, it invokes that function, rather than returning the function itself.

Usually that is exactly what is wanted, as in the example above. But occasionally you may want to copy the function reference, rather than invoking the function. You can do this by prepending your identifier with an @ (read "address of"). Example:

```
triple = function(n=1)
  return n*3
end function
x = @triple
print(x(5))     // prints: 15
```

Here we've again declared a function and stored it in a variable called `triple`. Then we copy the *address of* that function into another variable called `x`. At this point we can invoke the function either way, via `triple` or via `x`, and both do exactly the same thing. Had we left out the @ on the assignment, MiniScript would have instead evaluated the function `triple` refers to, and assigned the result (3) to `x`.

Here's a more realistic example. We'll define a function called `apply` which can apply a given function to each element of a list. Then we can invoke this on a list with any function, simply by using @ to refer to the function we want to apply.

```
apply = function(list, func)
  result = list[0:]                // make a copy of the list
  for i in indexes(result)
    result[i] = func(result[i])    // apply func to each element
  end for
  return result                    // return modified result
end function

print(apply([1, 2, 3], @triple))   // prints: [3, 6, 9]
```

To summarize, you invoke a function by simply using any identifier that refers to it. You avoid this invocation, and refer instead to the function itself, by putting @ before the identifier.

Classes and Objects

MiniScript supports object-oriented programming (*OOP*) via prototype-based inheritance. That is, there is fundamentally no difference in MiniScript between a class and an object; the difference, when it exists at all, lives solely in the intent of the programmer.

A class or object is a map with a special `__isa` entry that points to the parent (prototype). This is an implementation detail you rarely need to worry about, because it is handled automatically by the following rules:

1. When you create a map using the special `new` operator, the `__isa` member is set for you.
2. When you look up an identifier in a map, MiniScript will walk the `__isa` chain looking for a map containing that identifier. The value returned is the first value found.

These simple rules provide everything needed for object-oriented programming. A series of "classes" may be defined as maps containing functions and default data, which are inherited or overridden as needed. An "object" is just another map, inherited from some class, which normally contains only custom data.

Let's illustrate with an example. We'll define a class called Shape, with a subclass called Square.

```
Shape = {}
Shape.sides = 0

Square = new Shape
Square.sides = 4
```

A base class is just an ordinary map; in this case, we added a `sides` entry with a value of 0, signifying that "sides" is a bit of data we expect every Shape to have. Then we created a subclass by using `new Shape`, and assigned this to Square. In Square, we overrode the value of `sides` (as all squares should have 4 sides).

Now let's create an instance of our Square class, again by using `new`.

```
x = new Square
print(x.sides)     // prints: 4
```

Notice how we're using the traditional OOP terminology of "class" and "instance" for convenience, but in reality, there are just three maps — Shape is the prototype of Square, and Square is the prototype of x. The `__isa` member of each map points to the prototype, because we created them with `new`.

Now let's add a function to the Shape class, which should work for any shape subclass or object.

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

print(x.degrees)    // prints: 360
```

This example illustrates one additional rule important to object-oriented programming:

- When a function is invoked via dot indexing, it receives a special `self` variable that refers to the object on which it was invoked.

So in the example above, we invoked the `degrees` function as `x.degrees`, which looks for a member called "degrees" in `x` (and its prototypes via the `__isa` chain). And when that function is invoked, a special local variable called `self` is bound to the `x` object, i.e. the first map in the search chain. This allows class functions to access object data.

Intrinsic Functions

built-in functions you can rely on

MiniScript comes with a standard set of built-in (or *intrinsic*) functions. Many of these are globals (i.e., referred to by variables in the global space). Others (particularly functions intended for use with strings, lists, and maps) are normally invoked via dot syntax after an identifier.

In fact, though, all intrinsic functions that use dot syntax are written in such a way that they can *also* be invoked as global functions. So, for example, you can get the length of a string `s` by typing `s.len`, but you can also do the same thing as `len(s)`.

The following tables list the standard intrinsic functions, divided by data type on which they operate. Keep in mind that MiniScript is intended to be embedded in some host environment, such as a game or application. The host will normally add additional intrinsic functions particular to that environment. Please consult the documentation or help materials for your host environment for information on these extra functions.

Numeric Functions

MiniScript includes a selection of trigonometric functions, which all work in radians (rather than degrees), and other math functions, as well as random numbers and conversion of numbers into strings.

In the following table, *x* is any number, *i* is an integer, and *r* is a number of radians.

abs(x)	absolute value of x
acos(x)	arccosine of x, in radians
asin(x)	arcsine of x, in radians
atan(x)	arctangent of x, in radians
ceil(x)	next whole number equal to our greater than x
char(i)	returns Unicode character with code point i
cos(r)	cosine of r radians
floor(x)	next whole number less than or equal to x
pi	3.14159265358979
range(x, y=0, step=null)	returns a list containing values from x through y, in increments of step; step == null is treated as a step of 1 if y > x, or -1 otherwise
round(x, d=0)	x rounded to d decimal places

rnd(seed=null)	if seed=null, returns random number in the range [0,1); if seed != null, seeds the random number generator with the given integer value
sign(x)	sign of x: -1 if x < 0; 0 if x == 0; 1 if x > 0
sin(r)	sine of r radians
sqrt(x)	square root of x
str(x)	converts x to a string
tan(r)	tangent of r radians

String Functions

All string functions except `slice` are designed to be invoked on strings using dot syntax, but can also be invoked as globals with the string passed in as the first parameter. Note that strings are immutable; all string functions return a *new* string, leaving the original string unchanged. In the following table, *self* refers to the string, *s* is another string argument, and *i* is an integer number.

.code	Unicode code point of first character of self
.hasIndex(i)	1 if i is in the range [0, self.len); otherwise 0
.indexes	range(0, self.len-1)
.indexOf(s, after=null)	0-based position of first substring s within self, or null if not found; optionally begins the search after the given position
.len	length (number of characters) of self
.lower	lowercase version of self
.remove(s)	self, but with first occurrence of substring s removed (if any)
.upper	uppercase version of self
.val	converts self to a number (if self is not a valid number, returns 0)
.values	list of individual characters in self (e.g. "spam".values = ["s", "p", "a", "m"]
slice(s, from, to)	equivalent to s[from:to]

List Functions

All list functions except `slice` are designed to be invoked on lists using dot syntax, but can also be invoked as globals with the list passed in as the first parameter. Lists are mutable; the `pop`, `pull`, `push`, `shuffle`, and `remove` functions modify the list in place. To use a list like a stack, add items with `push` and remove them with `pop`. To use a list like a queue, add items with `push` and remove them with `pull`.

In the following table, *self* is a list, *i* is an integer, and *x* is any value.

.hasIndex(i)	1 if i is in the range [0, self.len); otherwise 0
.indexes	range(0, self.len-1)
.indexOf(x, after=null)	0-based position of first element matching x in self, or null if not found; optionally begins the search after the given position
.len	length (number of elements) of self
.pop	removes and returns the last element of self (like a stack)
.pull	removes and returns the first element of self (like a queue)
.push(x)	appends the given value to the end of self; often used with pop or pull
.shuffle	randomly rearranges the elements of self (in place)
.sort(key=null)	sorts list in place, optionally by value of the given key (e.g. in a list of maps)
.sum	total of all numeric elements of self
.remove(i)	removes element at index i from self (in place)
slice(list, from, to)	equivalent to list[from:to]

Map Functions

Functions on maps are very similar to functions on lists. Maps (like lists) are mutable; the `push`, `pop`, `remove`, and `shuffle` methods modify the map in place. You can treat a map like a set using `push`, which inserts 1 (true) for the value of the given key, and `pop`, which returns a key and removes it (and its value) from the map. Keep in mind that the order of keys in a map is undefined.

In the following table, *self* is a map, *i* is an integer, and *x* is any value.

.hasIndex(x)	1 if x is a key contained in self; 0 otherwise
.indexes	list containing all keys of self, in arbitrary order
.indexOf(x, after=null)	first key in self that maps to x, or null if none; optionally begins the search after the given key
.len	length (number of key-value pairs) of self
.pop	remove and return an arbitrary key from self
.push(x)	equivalent to self[x] = 1
.remove(x)	removes the key-value pair where key=x from self (in place)

.shuffle	randomly remaps values for keys
.sum	total of all numeric values in self
.values	list containing all values of self, in arbitrary order

System Functions

The following functions relate to the operation of MiniScript itself, or interact with the host environment. The latter (print, time, and wait) are only quasi-standard, in that support for them depends on the host application, and so they may not function in some environments.

globals	reference to the global variable map
locals	reference to the local variable map for the current call frame
print(x)	convert x to a string and print to some text output stream
time	number of seconds since program execution began
wait(x=1)	wait x seconds before proceeding with the next MiniScript instruction
yield	wait for next invocation of main engine loop (e.g., next frame in a game)

Examples

small programs that do interesting things

While we've given short examples of MiniScript code throughout this manual, this chapter presents several longer, more interesting examples. Many of the tasks illustrated are taken from RosettaCode, an online database of programming challenges with solutions in many languages. You can go there to compare the MiniScript solution to any other language; you may be amazed how much more readable MiniScript is than the alternatives.

FizzBuzz

FizzBuzz is a standard introductory-level programming challenge[1]. The task is simple: print the numbers 1 through 100, *but:* for multiples of three, print "Fizz" instead of the number; for multiples of five, print "Buzz" instead of the number, and for any number that's a multiple of three *and* five, print "FizzBuzz".

There are clearly many ways to tackle this; here's one.

```
1. fizzBuzz = function(n)
2.   for i in range(1, n)
1.     s = "Fizz" * (i%3==0) + "Buzz" * (i%5==0)
2.     if s == "" then s = str(i)
3.     print(s)
4.   end for
5. end function
6. fizzBuzz(100)
```

Instead of just hard-coding a loop from 1 to 100, we've made a function that can FizzBuzz up to any number. Within that function, the only clever bit is line 3, which takes advantage of a couple of MiniScript features. First, comparisons (such as `i%3==0` — read "i mod 3 equals zero") evaluate to 1 when true, or 0 when false. Second, you can multiply a string by a number to repeat it that many times. This means that if you multiply a string by a condition, you get either the original string (if the condition is true) or the empty string (if it is false).

That lets us easily generate "Fizz", "Buzz", and "FizzBuzz" depending on what our loop counter is divisible by. Line 4 simply fills in the number if we don't get one of those strings. (Quiz: can you rewrite this line to use the same multiply-by-condition trick as line 3?)

[1] http://rosettacode.org/wiki/FizzBuzz

Filter

Here's another RosettaCode task[2]: select certain elements from an Array into a new Array in a generic way. To demonstrate, select all even numbers from an Array.

```
filter = function(seq, f)  // filter seq to where f is true
  result = []
  for i in seq
    if f(i) then result = result + [i]
  end for
  return result
end function

isEven = function(x)
  return x % 2 == 0
end function

list = [2,3,5,6,8,9]
print(filter(list, @isEven))
```

This is a pretty straightforward conversion of the task description into MiniScript code. Our `filter` function takes a list and a function, and builds a new list by appending each element where the function, applied to that element, is true.

We illustrate by making an `isEven` function that returns true only when its argument mod 2 is zero (i.e., the argument is evenly divisible by 2). Then we pass `@isEven` to find just the even elements of a given list.

Greatest Common Divisor

Here's a function that finds the biggest number that can divide evenly into two given numbers[3]. Middle schoolers everywhere will soon be out of work.

```
gcd = function(a, b)
  if a == 0 then return b
  while b != 0
    newA = b
    b = a % b
    a = newA
  end while
  return abs(a)
end function
print(gcd(-21, 35))
```

The algorithm here, known as the "Euclidian algorithm for finding the GCD," is clever. The actual MiniScript code is simple.

[2] http://rosettacode.org/wiki/Filter

[3] http://rosettacode.org/wiki/Greatest_common_divisor

Maximum Element

MiniScript does not have a standard intrinsic for finding the maximum element of a list. But you can easily add it yourself, using this code.

```
1. max = function(seq)
2.   if seq.len == 0 then return null
3.   max = seq[0]
4.   for item in seq
5.        if item > max then max = item
6.   end for
7.   return max
8. end function
9. print(max([5, -2, 12, 7, 0]))
```

Pretty simple stuff. Line 2 checks to make sure the sneaky user hasn't given us an empty list; if they have, we return null, as there is no sensible max in that case. Otherwise, we just suppose it's the first element, and then loop over each element in the list, keeping the biggest.

Notice that the "max" variable assigned to on line 1 is in the global variable space, while the "max" assigned on lines 3 and 5 (and then returned on line 7) is local to a function. These happen to have the same name, but have nothing to do with each other. As a matter of style, it might have been better to name the local variable "result" rather than "max". But it seemed like a good opportunity to demonstrate how local and global variables are separate, even if they have the same name.

Titlecase

MiniScript has intrinsics to convert a string to all upper- or lower-case letters. But what if you want to capitalize just the first letter of each word, and lowercase the rest?

```
titlecase = function(s)
  result = ""
  for i in s.indexes
       if i == 0 or s[i-1] == " " then
            result = result + s[i].upper
       else
            result = result + s[i].lower
       end if
  end for
  return result
end function
print(titlecase(“SO LONG and thanks for all the fish"))
```

We just iterate over the string, capitalizing each letter than is either the very first character in the string, or is preceded by a space.

String Split

Splitting a string into a list of strings is an insanely useful thing to do. Alas, MiniScript doesn't have a standard intrinsic for that. Perhaps it will one day. Until then, here's code that does the job.

```
split = function(s, delim=" ")
  result = []
  wordStart = 0
  sLen = s.len
  delimLen = delim.len
  i = 0
  while i < sLen
    if s[i:i+delimLen] == delim then
      result = result + [ s[wordStart:i] ]
      wordStart = i + delimLen
      i = wordStart
    else
      i = i + 1
    end if
  end while
  return result + [ s[wordStart:i] ]
end function

print(split("Eat, pray, love", ", "))
print(split("...foo...bar...", "..."))
```

This loops over the string, looking for the delimiter, which can be either a single character or a longer substring. When we find it, we want to not only pull off the word up to that point, but also skip ahead and start looking at the string again after that delimiter. You can't easily jump ahead in a for loop, so we're using a while loop here instead. (While loops will let you do whatever you want; they just don't care.)

Notice how for performance, we copy `s.len` and `delim.len` into some local variables. That makes the code a little longer, but a little faster, since MiniScript doesn't have to call a function every time we reference these within the loop.