2021 International STEM Education Conference (iSTEM-Ed 2021)
November 10-12, 2021, Pattaya, THAILAND

MiniScript: A New Language for Computer Programming Education

Joseph Strout
Luminary Apps, LLC
joe@LuminaryApps.com

***Abstract*—Computer programming is fundamental to modern STEM (science, technology, engineering, & math) education and industry. Visual programming environments such as Scratch provide a gentle introduction to the topic for young children, but the transition to text-based languages such as Python or C# can be a difficult leap. A new scripting language, MiniScript, has been designed to fill this gap. It uses minimal syntax and a carefully selected small set of language features to present a less intimidating challenge for the learner. At the same time, the language is complete enough to be used for sophisticated games and other programs. Finally, MiniScript itself is lightweight and designed to be embedded into other software written in either C# or C/C++, making it more likely that children will encounter the language in the context of games, thus providing self-motivated learning opportunities.**

Keywords—programming, scripting, education, embedded languages

I. Introduction

Computer programming, or more generally computational thinking, is a foundational skill in science, technology, engineering, and math (STEM) fields. For young children, studies have shown that visual languages such as Scratch provide an effective introduction to programming skill. However, when the learner is ready to move beyond visual languages and start working with text-based programming languages, it may be a difficult leap.

The ideal first text-based language would have the following characteristics:

1) *Simplicity:* a language designed with minimal syntax, a minimal but complete set of flow-control constructs, and a small set of intrinsic functions (all features of visual languages like Scratch).

2) *Expressiveness:* the language should support all common software paradigms seen in production languages, including recursion, object-oriented programming, first-class functions, and proper handling of Unicode text.

3) *Ubiquity:* the language should appear in many contexts that may be intrinsically motivating to the user, for example, as an in-game or modding language in video games.

The first and third points are somewhat related; an easily embeddable language is likely to be small, and a small language is more likely (though not certain) to be simple. The second point is somewhat in opposition to the first two; simple, easily embedded languages tend to be less expressive than large, complex ones.

This paper introduces a new programming language designed to meet all three criteria. The language, MiniScript, is a small language designed for learning and teaching, with implementations optimized for embedding in C# or C++ games.

II. Related Work

MiniScript has been influenced by many predecessor languages, but two languages present particularly important comparisons: Python, for its elegant syntax and handling of both lists and maps; and Lua, for its small size and ease of embedding.

Python, first released in 1991, has become popular as both a production language and a teaching language [1]. Studies such as [2] have shown that novice students learn better with a simple language such as Python, than with a more complex one like Java. Like MiniScript, Python is a dynamically-typed language with a well-developed read/eval/print loop (REPL), and it features sophisticated handling of both lists (ordered sequences of values) and maps (sets of key/value pairs). However, at over 661 thousand lines in 718 source files, building Python can be challenging [3]; and it is not designed for embedding in other software (though doing so is not impossible). Python has gone through three major revisions, with a difficult transition from version 2 to 3 resulting in many users failing to upgrade or replace deprecated APIs [4].

Lua was first released in 1993, and has become a popular language for embedding in other software, especially games [5]. More than an order of magnitude smaller than Python, Lua was designed from the beginning as an extension language, though recent versions include a REPL as well [6]. Lua uses one data type, *table*, to represent both sequences and maps, leading to some common errors [7]; and it lacks the slice semantics commonly used in Python. Also notable in Lua is that all variables are global by default (a special `local` keyword is used to declare a local variable).

978-0-7381-1380-7/21/$31.00 ©2021 IEEE

MiniScript shares many features with these earlier languages, but has some important differences too. In handling of lists and maps, it is most similar to Python; but instead of indentation-based code blocks it uses block keywords, similar to Lua. While both Python and Lua predate widespread adoption of Unicode, MiniScript was created with Unicode support throughout. In terms of size, MiniScript is even smaller than Lua (see Table 1). Finally, while Python and Lua are both written in C, MiniScript comes with two functionally equivalent reference implementations: one in lightweight C++ (eschewing the Standard Template Library) and one in C#. This enables MiniScript to be easily embedded even in applications written in C#, such as games using the popular Unity engine.

III. Language Features

In this section, an overview of the language is provided. The purpose of this is to give the reader a quick sense of the style and capabilities of MiniScript. For a more complete description, please refer to the MiniScript manual.

MiniScript is a procedural, object-oriented, prototype-based language. Compared to most other programming languages, MiniScript uses relatively little punctuation, particularly in basic control flow statements; it needs no parentheses around the condition in `if` or `while` statements, for example, and code blocks are delimited by keywords rather than curly braces or parentheses. (See Listing 1.)

Mathematical operators are the same ones common to most modern languages: `+`, `-`, `*`, and `/`, plus `%` for mod and `^` for exponentiation. The logical operators are keywords `and`, `or`, and `not`; comparison operators are as in C or Python (`==`, `!=`, `>=`, etc.). Operator precedence is in standard algebra, with parentheses used only as needed for grouping subexpressions.

```
// print a countdown
for i in range(3,1)
    print "Ready in " + i
    wait
end for

// pick a random number
num = round(100 * rnd)
// loop until input is correct
while true
    x = input("Your guess?").val
    if x == num then
        print "Correct!"
        break // exit loop
    else if x > num then
        print "Too high."
    else
        print "Too low."
    end if
end while
```

Listing 1. A sample MiniScript program.

TABLE 1. Comparisons with Python and Lua.

	Python	Lua	MiniScript
Source Lines	661,775	29,469	13,752
Source Files	718	62	46
Data Types	34	8	6
Intrinsics	69	242	53

Table 1. Comparison of Python, Lua, and MiniScript on four measures of size: C/C++ source code lines; source code files; number of data types; and number of standard intrinsic functions.

Like Python 2, MiniScript does not need parentheses around the argument to the `print` statement. That feature was lost in Python 3, in order to make `print` more consistent with other functions. In MiniScript, omitting the parentheses is already consistent; they can (and should) be omitted any time a function call is the statement itself, rather than part of some larger expression. They are also omitted any time the argument list is empty. For example, consider:

```
print ceil(rnd * 10)
```

In this complete MiniScript statement, `print`, `ceil`, and `rnd` are all functions, but only `ceil` requires parentheses (because it takes arguments but is not the statement root). The result of this policy is clean, syntax-light code characterized by very little punctuation. It also makes the difference between computed and stored properties an implementation detail, rather than something that concerns the users of a class or module.

MiniScript uses three control flow constructs: `if/else`, `while`, and `for`. It also supports `continue` and `break` for skipping to the next iteration or jumping out of a loop.

MiniScript has exactly six data types:

1) *Numbers* are stored in full-precision format, and are also used to store `true` (1) and `false` (0).

2) *Strings* are immutable runs of Unicode characters, and support both iteration and slicing.

3) *Lists* reference mutable, ordered sequences of arbitrary values. Like strings, lists support iteration and slicing, but (unlike strings) lists can be modified in place.

4) *Maps* reference mutable collections of key/value pairs. Keys within a map are unique; both keys and values may be any type. Maps also form the basis of the class/object system, and support single inheritance.

5) *Functions* reference compiled subprograms. Functions in MiniScript are first-class objects and may be stored in variables, passed as parameters, etc.

6) *Null* is a special data type with only one value (the constant `null`).

Examples of all six types are shown in Table 2.

The MiniScript core contains 53 intrinsic methods, many of which are overloaded to work on multiple

TABLE 2. MINISCRIPT DATA TYPES (WITH EXAMPLES).

Data Type	Sample Usage
Number	`e = 2.718`
String	`s = "Hello World"`
List	`seq = [1, 2, "three"]` `s[0] = "first"`

Map	`m = {five:5}`
	`m["six"] = 6`
	`m.seven = 7`

Function	`dist = function(x,y)`
	`  return sqrt(x^2 + y^2)`
	`end function`

Null	`x = null`

types. For example, the `len` intrinsic will return the number of Unicode characters in a string, the number of elements in a list, or the number of key/value pairs in a map. The intrinsics support use of the core data types as many of the common data structures in computer science: e.g., to use a list as a stack or a queue, or to use a map as a set.

Indexing and slice syntax are very similar to Python: an element of a list or string is obtained by placing the index within square brackets after the list or string reference, and a sublist or substring results from specifying a range via a colon. For example, if `s` is a string, then `s[3]` returns character 3 (counting the first character as 0), and `s[3:5]` returns the substring from character 3 up to but not including character 5. Either index may be negative, in which case it counts backwards from the end of the string; or omitted, in which case the range implicitly starts or ends at the beginning or end of the string, respectively. Thus, s[-3:] returns the last three characters of a string. Lists work in exactly the same way, with the additional feature that elements (but currently not ranges) may be assigned new values, mutating the list.

Map syntax is inspired by Python as well; a map literal uses key:value pairs separated by commas, and enclosed in curly braces. Once a map is created, keys may be specified in square brackets in a manner very similar to indexing into a list or string. However, MiniScript maps support an additional feature: any string key that is a valid identifier may also be accessed via dot syntax, i.e. a map reference, followed by a dot operator (period), and then the key. This is mostly equivalent to the normal square-bracket syntax, unless the value of associated with the key is a function reference (more on this later). This dot notation is often a convenient alternative to square-bracket indexing, and helps prepare the learner for similar syntax in other languages; it also has additional semantics when referencing a function or in the presence of inheritance, as described below.

Functions are unnamed, first-class objects that can take any number of parameters (with default values), and return a single result value (which may be null). Variables within a function are always local by default, but function code may read variables in the calling scope or in the global scope as well. Assigning to values in these higher scopes can be done via the `outer` and `global` keywords. This default-local scoping is common in modern languages (though differs from Lua), and helps avoid a proliferation of global variables.

Functions are invoked by evaluating any variable that refers to them. This works both for simple variables, like `f`, as well as map properties referenced via dot syntax, such as `m.f`. When a function is invoked via dot syntax, it gets an implicit argument `self` that refers to the map on which it was invoked. Note that unlike Python, this `self` parameter is not part of the function parameter list; it is inserted implicitly by the invocation. Listing 2 illustrates the use of maps and functions.

Several additional features support object-oriented programming (OOP) via prototype-based inheritance. A map can be made to derive from a base map by creating it with the `new` operator, which sets a special "__isa" key. When the dot operator is evaluating the key (i.e. the right-hand side identifier), it will walk this `__isa` chain until a match is found. This allows derived maps to be used as subclasses or instances, overriding only the functions or other values needed,

```
Shape = {"sides": 0, "color": "blue"}
Shape.degrees = function()
    return 180 * (self.sides - 2)
end function
Shape.name = function()
    return self.sides + "-sided shape"
end function
Shape.describe = function(caps=false)
    s = "a "+self.color+" "+self.name
    if caps then s = s.upper
    print s
end function
Shape.sides = 3
print Shape.degrees
Shape.describe true
```

OUTPUT:

```
180
A BLUE 3-SIDED SHAPE
```

Listing 2. Illustration of map, function, and dot syntax.

```
Square = new Shape
Square.sides = 4
Square.name = "square"
Square.describe =
function(capitalize=false)
    super.describe capitalize
    print " (my favorite shape)"
end function

mySquare = new Square
mySquare.color = "yellow"

print mySquare.degrees
mySquare.describe
```

OUTPUT:

```
360
a yellow square
 (my favorite shape)
```

Listing 3. Subclassing and instantiation. (Append to Listing 2.)

```
commonPrefix = function(strList)
    if not strList then return null
    // find the shortest and longest strings (without sorting)
    shortest = strList[0]
    longest = strList[0]
    for s in strList
        if s.len < shortest.len then shortest = s
        if s.len > longest.len then longest = s
    end for
    if shortest.len < 1 then return ""
    // now find how much of the shortest matches the longest
    for i in range(0, shortest.len-1)
        if shortest[i] != longest[i] then return shortest[:i]
    end for
    return shortest
end function

items = ["interspecies", "interstellar", "interstate"]
print commonPrefix(items)
```

Listing 4. Function to find the longest common prefix of a list of strings.

and inheriting the rest from the base map. Finally, within a function, a special `super` keyword allows reference to the base map of the map on which the function was found; this supports the common pattern of invoking a base-class method from within derived-class code. Listing 3, which should be read as an extension of Listing 2, illustrates some of these OOP features.

This overview concludes with two more realistic examples, taken from the Rosetta Code website [8]. Listing 4 finds the longest common prefix of a list of strings; and Listing 5 shows both iterative and recursive methods to find a Fibonacci number.

IV. Conclusion

MiniScript is poised to become a useful new alternative for a language used to teach programming.

```
// Fibonacci number (recursive)
rfib = function(n)
    if n < 1 then return 0
    if n == 1 then return 1
    return rfib(n-1) + rfib(n-2)
end function

// Fibonacci number (iterative)
ifib = function(n)
    if n < 2 then return n
    n1 = 0
    n2 = 1
    for i in range(n-1, 1)
        ans = n1 + n2
        n1 = n2
        n2 = ans
    end for
    return ans
end function

print "Recursive: " + rfib(6)
print "Iterative: " + ifib(6)
```

Listing 5. Fibonacci number, with recursion and iteration.

Its minimal syntax, combined with modern features such as inheritance, first-class functions, default-local variables, and Unicode support, make it an appealing choice for both teaching and learning.

The primary drawback to MiniScript is that, as a very new language, it is not yet widely known or used. However, with light-weight, open-source implementations in both C++ and C# — the languages used by the highly popular Unreal and Unity game engines — there is reason to believe this may change in the future.

Acknowledgment

The author wishes to thank Georg Becker for helpful coments on the manuscript.

References

1. C. S. Miller, A. Settle, and J. Lalor, “Learning object-oriented programming in Python: Towards an inventory of difficulties and testing pitfalls,” Proceedings of the 16th Annual Conference on Information Technology Education, pp. 59-64, 2015.
2. L. Mannila, M. Peltomäki, and T. Salakoski, “What about a simple language? Analyzing the difficulties in learning to program,” Computer Science Education, vol.16, no. 3, pp. 211-227, 2006.
3. V. Boykis, “It's still hard for beginners to get started with Python,” http://veekaybee.github.io/2018/03/12/installing-python-is-hard , 2018.
4. J. Wang, L. Li, K. Liu, and H. Cai, “Exploring how deprecated Python library APIs are (not) handled,” Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 233-244, 2020.
5. R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho, “Lua —an extensible extension language,” Software: Practice and Experience, vol. 26, no. 6, pp. 635-652, 1996.
6. R. Ierusalimschy, L. H. De Figueiredo, and C. Waldemar, Lua 5.1 reference manual, 2006.
7. “Avoiding gaps in tables used as arrays,” https://riptutorial.com/lua/example/8360, retrieved May 2020.
8. “Category: MiniScript”, http://www.rosettacode.org/wiki/Category:MiniScript, retrieved May 2020.