Lytescript (Revision 1) Specification

# Abstract

Lytescript (also referred to as Lyte) is a stack-based, multi-paradigm programming language that strives to have as little syntax as possible while still maintaining flexibility. With its “batteries-included” philosophy towards its standard library, this means that the language is easy to learn but powerful nevertheless. Lyte is primarily a mixture of the functional, loosely-typed and object-oriented programming language paradigms each of which affect the language differently. For example, it considers everything to be an object and reduces all arithmetic operations to function-calls.

# Syntax Overview

## In-line Functions

{ /\* Function Body \*/ }

@( /\* Arguments \*/ ) { /\* Function Body \*/ }

Lytescript provides two ways to create in-line functions, also called blocks, which simply create a new block and push it onto the stack. The first of these is simply the function’s body surrounded by curly-brackets whereas the second method denotes a set of named arguments. The latter is especially useful whenever arguments that would normally be lost after being popped off of the stack need to be used multiple times.

## Function Invocation

F(a, b) // Traditional  
b a F // Stack-based  
b `F` a // Infix notation

Lytescript provides multiple ways to invoke a function for the sake of code readability. The first of which is the “traditional” method which “invisibly” wraps each of its arguments in blocks and pushes them onto the stack in reverse-order. The stack-based method is the simplest way to invoke a function, as it simply assumes that the programmer has prepared the arguments on the stack and invokes the function. Lastly, using infix notation swaps the order of execution of the statement surrounded by back ticks and the following statement. Because of these properties, all three of the above examples are equivalent. Note, the infix method does not imply anything about the order of operations besides the aforementioned swapping; the limitation of this can be seen in the example below.

4 `+` 2 `\*` 6 /\* This results in 36 rather than the result accounting for  
 the order of operations, 16 \*/

## Primitives

### Numbers & Strings

4 4.0 0b100 0x4 04 4E0  
"An example string"

Lytescript provides several ways to represent a number, from right to left (in the above example) integer, floating point, binary, hexadecimal, octal and scientific. Each of these will push a representative Number object onto the stack. Strings are single-line chunks of text surrounded by double-quotes that allow for Javascript-style escapes.

### Objects

%{ Key: Value, Key: Value, ... }

Objects are sets of key-value pairs (aka maps) that allow for keys that are valid identifiers, numbers or strings while values can be of any type. For more information about valid keys and different kinds of properties see Working with Objects.

### Lists

%[ (Value | Range), (Value | Range), ... ]

Lists are integer-indexed collections of values of any type and ranges.

### Ranges

Start:Finish  
Start:Step:Finish

Ranges are lists of numbers over that step by 1, -1 or a given amount over a given range (inclusive). They can only be created within Lists.

## Assignment Operations

**Destination** <- value  
-> **Destination**  
-> [**First**, **Second**, **Third**, ...]

Like infix notation, the first of these assignment operations swaps the order of execution for readability purposes (in this case pushing the value onto the stack before popping it into the destination, it is important to note this only swaps the order of one statement so things like i <- i `+` 1 will produce unexpected results). This syntax is typically used to initialize objects, templates or global functions. The second method simply pops a value off of the stack into the destination and should generally be used for updating/setting any other values. Finally, the third method binds to one (or more) destinations, popping the values into their destinations in left-to-right order.

## Accessing List Elements and Object Properties

object[key/index] *–or-* list[index]  
object.key *–or-* list.key  
#key *–or-* @key  
#[index/key] *–or-* @[index/key]

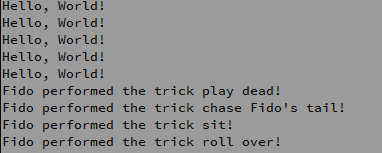
Lytescript provides a few different ways to index an object/list, the first two being fairly self-explanatory (as they mimic the behavior of many other languages). The latter two provide a way to access a specific object, the # prefix accesses the object on top of the stack whereas the @ prefix accesses the object set as the scope’s “self.”

## A Brief Example/Recap

**HelloWorld** <- @(times) {  
 **If**(times `>` 1, HelloWorld(times `-` 1), {})  
 **EchoLn**("Hello, World!")  
}

HelloWorld(5)

**Fido** <- %{  
 name: "Fido",  
 tricks: %[  
 "play dead",  
 "chase " `++` @name `++` "'s tail",  
 "sit",  
 "roll over"  
 ],  
 doTricks: {  
 **For**(@tricks, @(trick) {  
 **EchoLn**(@name `++` " performed the trick " `++` trick `++` "!")  
 })  
 }  
}  
  
Fido.doTricks()

Upon first looking at this example, one might be confused by the sudden appearance of things that resemble keywords. These “keywords” are actually functions from within Lyte’s Standard Library. Based on its output, it’s pretty easy to see the HelloWorld function just recursively prints a hello world message. Next, we can see that Fido is an object with a list of tricks. When the doTricks function is invoked, it iterates through this list and “performs” each of the tricks.

# Standard Library Overview

Lyte’s standard library is made up of several packages that provide over 100 different functions. For ease of use, some of these functions are accessible via aliases in the global scope (such as Lyte.Math.Add which has the alias +).

## The Core Package

|  |  |  |
| --- | --- | --- |
| **Full Path** | **Alias** | **Brief** |
| **Lyte.Core.And** | And | Computes the logical and of two values |
| **Lyte.Core.Apply** | Apply | Applies the block on the stack |
| **Lyte.Core.BitwiseAnd** |  | Computes the bitwise and of two numerical values |
| **Lyte.Core.BitwiseNot** |  | Computes the bitwise not of a numerical value |
| **Lyte.Core.BitwiseOr** |  | Computes the bitwise or of two numerical values |
| **Lyte.Core.BitwiseXor** |  | Computes the bitwise xor of two numerical values |
| **Lyte.Core.Dig** | Dig | Digs up the nth value to the top of the stack (Dig(2) is equivalent to Swap) |
| **Lyte.Core.Equal** | == | Tests if two values are unequal, coercing their types if needed |
| **Lyte.Core.EqualStrict** | === | Tests if two values are strictly equal, in other words if they are of the same type and value |
| **Lyte.Core.For** | For | Takes either a list or two numbers and a function, iterates through the values in the range (inclusive) or through each element in a list, pushing the current value onto the stack then calling the function each time |
| **Lyte.Core.If** | If | If the first argument is truthy execute the second argument otherwise execute the third |
| **Lyte.Core.Import** | Import | Imports an external class, see Importing Classes & Packages for more details. |
| **Lyte.Core.IsNull** | Null? | Checks to see if a value is null |
| **Lyte.Core.IsUndefined** | Undefined? | Checks to see if a value is null |
| **Lyte.Core.Not** | Not | Computes the logical not of a value. |
| **Lyte.Core.NotEqual** | != | Tests if two values are equal, coercing their types if needed |
| **Lyte.Core.NotEqualStrict** | !== | Tests if two values are strictly unequal, in other words if they are of the same type and value |
| **Lyte.Core.Null** | Null | Returns the null value |
| **Lyte.Core.Or** | Or | Computes the logical or of two values |
| **Lyte.Core.Pop** | Pop | Pops the value on top of the stack |
| **Lyte.Core.Same** | Same? | Checks if two values are references of the same object. |
| **Lyte.Core.Swap** | Swap | Swaps the top two values on the stack. |
| **Lyte.Core.ToBool** | ToBool | Coerces the value on top of the stack into a Boolean |
| **Lyte.Core.ToNumber** | ToNumber | Coerces the value on top of the stack into a Number |
| **Lyte.Core.ToString** | ToString | Coerces the value on top of the stack into a String |
| **Lyte.Core.TypeOf** | Type? | Returns the type of the value on top of the stack (popping it in the process) |
| **Lyte.Core.Undefined** | Undefined | Returns the undefined value |
| **Lyte.Core.Unless** | Unless | Unless the return argument is truthy invoke the second argument otherwise execute the third |
| **Lyte.Core.Until** | Until | Until the return value of the first argument is truthy, invoke the second argument |
| **Lyte.Core.Version** |  | Returns the Current Version of Lytescript you are using (as a String) |
| **Lyte.Core.While** | While | While the return value of the first argument is truthy, invoke the second argument |

## The Math Package

Note: The Math package is pre-imported into the global scope eliminating the need for an **Import**("Lyte.Math") call.

## The Testing Package

|  |  |  |
| --- | --- | --- |
| **Full Path** | **Alias** | **Brief** |
| **Lyte.Test.AssertDefined** |  | Asserts that a value is defined |
| **Lyte.Test.AssertEquals** |  | Asserts that two values are equal |
| **Lyte.Test.AssertEqualsStrict** |  | Asserts that two values are strictly equal |
| **Lyte.Test.AssertFalse** |  | Asserts that a value is falsey |
| **Lyte.Test.AssertNotEquals** |  | Asserts two values are not equal |
| **Lyte.Test.AssertNotEqualsStrict** |  | Asserts two values are not strictly equal |
| **Lyte.Test.AssertNotNull** |  | Asserts that a value is not null |
| **Lyte.Test.AssertDifferent** |  | Asserts that two values are references to different objects |
| **Lyte.Test.AssertNull** |  | Asserts that a value is null |
| **Lyte.Test.AssertRaises** |  | Asserts that a block raises an error |
| **Lyte.Test.AssertSame** |  | Asserts that two values are references to the same object |
| **Lyte.Test.AssertTrue** |  | Asserts that a value is truthy |
| **Lyte.Test.AssertUndefined** |  | Asserts that a value is undefined |
| **Lyte.Test.Fail** |  | Causes a test to fail |
| **Lyte.Test.Test** |  | Runs a set of tests, See Unit Testing for more details. |

## The System Package

|  |  |  |
| --- | --- | --- |
| **Full Path** | **Alias** | **Brief** |
| **Lyte.System.Beep** |  | Beeps with a given frequency for a given duration (ms) |
| **Lyte.System.CurrentDirectory** |  | Gets the current system directory |
| **Lyte.System.Execute** |  | Execute a shell command |
| **Lyte.System.Exit** |  | Exit the VM completely (with an optional integer argument for the return code) |
| **Lyte.System.PathSeperator** |  | A platform-specific separator character used to separate path strings in environment variables. |
| **Lyte.System.Platform** |  | Gets the current platform identifier and version number. |
| **Lyte.System.Seperator** |  | Provides a platform-specific character used to separate directory levels in a path string that reflects a hierarchical file system organization. |

## The Utilities Package

|  |  |  |
| --- | --- | --- |
| **Full Path** | **Alias** | **Brief** |
| **Lyte.Util.CharToInt** |  | Converts a Character to its numerical, Unicode representation. |
| **Lyte.Util.Concatenate** | ++ | Concatenates two lists resulting in a third, new value |
| **Lyte.Util.EscapeString** |  | Escapes any special characters within a string back to their “safe” representation. |
| **Lyte.Util.Instantiate** | Instantiate | Instantiates an object template, calling its \_\_constructor. For more details see Template & Mixins. |
| **Lyte.Util.IntToChar** |  | Converts a numerical, Unicode value to its Character representation. |
| **Lyte.Util.IsMixedWith** | MixedWith? | Checks to see if an object is mixed with a mixin. For more details see Template & Mixins. |
| **Lyte.Util.MakeList** | MakeList | Makes a list from *all* of the values presently on the stack. |
| **Lyte.Util.MixWith** | MixWith | Mixes an object with a mixin. For more details see Template & Mixins. |
| **Lyte.Util.UnescapeString** |  | Unescapes any special characters within a string. |

## The Error Package

|  |  |  |
| --- | --- | --- |
| **Full Path** | **Alias** | **Brief** |
| **Lyte.Error.Raise** | Raise | Raises a value as an error |
| **Lyte.Error.Try** | Try | Tries to invoke the first block on the stack but if an error is raised by it invokes the second block, pushing the raised error object onto the stack. |

## The Reflection Package

|  |  |  |
| --- | --- | --- |
| **Full Path** | **Alias** | **Brief** |
| **~~Lyte.Reflect.Clone~~** |  | *Removed from Revision 1 of the Specification* |
| **~~Lyte.Reflect.DeepClone~~** |  | *Removed from Revision 1 of the Specification* |
| **Lyte.Reflect.Finalize** | Finalize | Finalizes a variable (so that it cannot be changed) |
| **Lyte.Reflect.Get** |  | Resolves the raw value of a variable (without applying it) |
| **Lyte.Reflect.GetProperties** |  | Gets a list of the properties of a value |

## The Input/output Package

|  |  |  |
| --- | --- | --- |
| **Full Path** | **Alias** | **Brief** |
| **Lyte.IO.Echo** | Echo | Writes a string to StdOut or another stream (if given) |
| **Lyte.IO.EchoLn** | EchoLn | Writes a string to StdOut or another stream (if given) with an endline character |
| **Lyte.IO.OpenFile** | OpenFile | Opens a file (whose path is given by the first argument) in the configuration given by the second argument. See Inputs & Outputs. |
| **Lyte.IO.Read** | Read | Reads a string from StdIn |
| **Lyte.IO.ReadLn** | ReadLn | Reads a line from StdIn |
| **Lyte.IO.StdIn** | StdIn | Returns the Standard Input Stream |
| **Lyte.IO.StdErr** | StdErr | Returns the Standard Error Stream |
| **Lyte.IO.StdOut** | StdOut | Returns the Standard Output Stream |

# Working with Objects

## A Word of Caution

In Lytescript, the underlying type for all keys is string which means one should exercise caution when using numeric strings with trailing zeros as a key (such as in the case of obj[“1.0”]). This is because if one coerces a number into a string it would not have the trailing zeros and consequently would not be a valid key for the object.

## Computed Property Names

%{  
 [*Statements*]: Value  
 }

Lytescript allows for computed property names at the time of an object’s initialization by simply wrapping the statements in brackets.

## Getters/Setters

%{  
 <- *Getter*: *Block*,  
 -> *Setter*: *Block*  
 }

At the time of an object’s initialization, one can override the default getting/setting behavior of a property using a getter/setter pair. Instead of getting/setting the objects properties directly, Lyte will invoke the block. In the case of a Getter it expects the block to push a value onto the stack whereas in the case of a Setter it expects the block to pop the new value off of the stack.

# Type Coercion

# Templates & Mixins

# Importing Classes & Packages

# Unit Testing