Coda Script Documentation

Coda, simply put, is a powerful scripting language that all Bethesda Game Studios Editor Extender plugins implement. It's primarily used to automate operations inside an editor environment. Tedious and monotonous tasks like renaming objects, renumbering form IDs, batch editing object attributes, etc, can be completed with just a few lines of code.

Salient Features

- Unlimited expression complexity.
- Supports floats, form references, strings and arrays.
- Supports user-defined functions.
- Supports control structures.
- Supports the background execution of scripts.
- Syntactically similar to Legacy/ObScript.
- High performance.

Syntax

Coda draws its inspiration from the scripting language Oblivion uses, Legacy. Its syntax derives directly from that of (OBSE) extended Legacy, in order to keep the learning curve shallow while allowing extensibility. There are, however, a few striking differences between the two.

- While Legacy implements explicit data types such as short, long, float, etc., *Coda* combines them all into a single variant for ease of use.
- Variable assignment is performed with just the necessary operators; special keywords such as **let** or **set** are not required.
- Script command calling syntax is similar to those of programming languages such as C++, Java, etc. – arguments are placed inside parentheses and delineated with commas.

Script Anatomy

A "typical" *Coda* script looks something like this:

```
CODA(SampleCodaScript, "0.01")

VAR aFirst
VAR aSecond = 2.7181812
```

```
VAR aThird = “This is a sample Coda script!”

VAR aArg1
VAR aArg2

BEGIN aArg1 aArg2
   PrintToConsole(“Hello World”)

   ; begin test

   aFirst = 45 * 8 – (2154 / 1.689)
   aSecond = (int)aArg1 + (float)aArg2

   IF (aFirst > aSecond)
      aThird = “Ding!”
      aFirst = 1
   ELSEIF (aFirst < aSecond)
      aThird = “Dong!”
      aSecond = 1
   ELSE
      aThird = “Duckling!”
      aArg1 = -1
   ENDIF

   aArg1 = 0
   aArg2 = 10
   aThird = ArrayCreate(10)

   IF (aArg1 == aArg2)
      RETURN(aArg1 * 353.555)
   ELSEIF (aArg1 == aArg2 * 12)
      RETURN(aSecond)
   ELSE
      aSecond = CALL(“UserFunction23”, \
                  aThird, \
                  “Calling a user-defined function!”)
   ENDIF

   WHILE (aArg1 < aArg2)
      ArrayInsert(aThird, aArg1, -1)
      aArg1 += 1

      IF (aArg1 == 3 || aArg1 == 5)
         CONTINUE()
      ELSEIF (aArg1 == 8)
         BREAK()
      ENDIF
   LOOP
```

```
   FOREACH aArg2 <- aThird
      PrintToConsole($aArg2)
   LOOP

   ; end test
END
```

Looks intimidating, doesn't it? Well, that sample script intends to showcase most of the language's features. A regular script would be much more simple, rest assured. *Coda* will be able to scale to most, if not all, of your needs.

A script can be broken down to the following components:

1. **Script Name** –
   Syntax:
   ```
   CODA(<script name> [, "polling interval"])
   ```

   As with Legacy, *Coda* scripts need to begin with a script name declaration. The name of the script must the same as the file name it's saved under. The polling interval is the duration in seconds between successive executions of the script when it's running in Background mode. It may be omitted, in which case it will be executed every time the script backgrounder updates.

2. **Variable Declarations** –
   Syntax:
   ```
   VAR <variable name> [= <initial value>]
   ```

   Variables are to be declared immediately following the script name declaration. There are no limits placed on the number of variables that can be declared inside a script. All declarations have local scope i.e., they can be accessed only by their parent script. However, all scripts can access the global variable pool. Variable initializers can be any complex script expression.

3. **Script Body** –
   Syntax:
   ```
   BEGIN [<parameters 1…10>]
         <code>
   END
   ```

   The BEGIN block encapsulates all of the script's executable code. The declaration can optionally contain a parameter list. Each parameter must correspond to a pre-declared variable, which will be initialized with the respective argument before the script's execution. Scripts are allowed to have up to 10 parameters. Note that parameters can only be passed to a script through the CALL command.

Scripts are case-insensitive. Comments can be added after a semicolon delimiter. Script expressions can be split into multiple lines by terminating each line with a single backslash (\).

Variables and Data Types

Coda supports the following data types:

- Numeric
- String
- Reference
- Array

Numeric values are double precision floating point numbers. Strings are a sequence of ASCII characters. Reference is a special type used to store forms or records (any object that can be created in the editor). All editor objects have a unique Form ID, which is what the reference data type tracks. Arrays are collections of data elements stored together, each identified by an integer called an index. Their elements can be of any supported data type, including other arrays. Array indices are zero-based i.e., they range from 0 to n-1 where n is the size of the array. Array variables are passed by reference and automatically garbage collected. Arrays and strings are manipulated using special commands (look into the command documentation).

Variables in *Coda* are variants – they can contain data of any of the supported types. However, they can only store one type at a time. For instance, they cannot contain an integral value and a string at the same time. The data type of a variable's value can be determined at run-time by the $(TYPEID)$ operator. The operator returns one of the following values indicating the variable's type:

- TYPEINFO_INVALID – The variable is uninitialized.
- TYPEINFO_NUMERIC – The variable contains a number.
- TYPEINFO_STRING – The variable contains a string.
- TYPEINFO_REFERENCE – The variable contains a reference.
- TYPEINFO_ARRAY – The variable contains an array.

Example:

```
IF ((TYPEID)SomeVariable == TYPEINFO_STRING)
   ; SomeVariable has a string value
ELSEIF ((TYPEID)SomeVariable == TYPEINFO_NUMERIC)
   ; numeric value
ENDIF
```

Script commands (and certain operators) often expect parameters of a particular type. Due to their polymorphic nature, variables sometimes need to be explicitly converted or cast into the required data type. Casting a variable forces the script engine to reinterpret its value as one of the cast type. The

following type-casts are supported by the engine (casting between the same types is supported implicitly):

- Number --> String
- Number --> Reference
- Reference --> Number
- Reference --> String
- Array --> String (size of the array)

The following type-casts are automatically performed by the engine whenever necessary:

- Reference --> Number

Casting operations are performed by using the various infix operators that define the destination data type. The following casting operators are supported by the engine:

- (NUM) - Cast to number.
- (REF) - Cast to reference.
- (ARRAY) - Cast to array, provided for completeness.
- (STR) or $ - Cast to string.

Examples:

```
VarA = (INT)VarB
VarA = (REF)VarB
VarB = (STR)VarC
PrintToConsole($VarA)
```

Operators

Coda implements a variety of mathematical, logical and relational operators for use in expressions. Precedence is the same as that in C++.

Standard Operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent

Assignment Operators:

=	Assignment
+=	Add and Assign. Adds the expression on the right to the variable/array element on the left.
-=	Subtract and Assign. Subtracts the expression on the right from the variable/array element on the left.
*=	Multiply and Assign.
/=	Divide and Assign.

Logical Operators:

AND *or* &&	Logical And. True if both expressions are true.
OR *or* ||	Logical Or. True if either expression is true.
XOR	Logical Xor. True if one expression is true and the other false.

Relational Operators:

==	Equality. True if the operands are equal. Operands must be comparable to each other.
!=	Inequality. True if the operands are not equal.
>	Greater Than.
<	Less Than.
>=	Greater or Equal.
<=	Less than or Equal.

Bitwise Operators:

&	Bitwise And.
|	Bitwise Or.
<<	Binary Left Shift.
>>	Binary Right Shift.

Miscellaneous Operators:

//	String Concatenation.
-	Sign Operator.
[]	Index/Subscript Operator. Used with arrays, returns a reference to the

	array element at a specific index.
	
	Example:
	
	`VarArray[3] = 234`
	`VarA = 43.1 / VarArray[0]`

Numeric Notation

Coda allows numbers to be represented as decimal (base 10), hexadecimal (base 16) or binary (base 2) values inside expressions.

- Hexadecimal – 0x0044FFA2, 0xBEEF
- Binary – #010010, #11011

Constants

The script parser supports the following constants:

- Pi – 3.14159265358979323846264 3
- E – 2.71828182845904523536028 7

Intrinsic Math Functions

All angles are represented as radians.

- Abs - Returns the absolute value of a number.
- Sin, Cos, Tan - Returns the sine, cosine and tangents respectively.
- SinH, CosH, TanH - Returns the hyperbolic sine, cosine and tangents respectively.
- ASin, ACos, ATan - Returns the arc sine, cosine and tangents respectively.
- ASinH, ACosH, ATanH - Returns the hyperbolic arc sine, cosine and tangents respectively.
- Log, Log10 - Returns the base-10 logarithm of a number.
- Log2 - Returns the base-2 logarithm of a number.
- Ln - Returns the natural logarithm of a number.
- Sqrt - Returns the square root of a number.
- Exp - Returns the exponential of a number i.e., e^{number}
- Min(x, y …z) - Returns the smallest number in a set.
- Max(x, y …z) - Returns the largest number in a set.

- Sum(x, y …z) - Returns the sum of all the numbers in a set.

Language Constructs & Features

As with all other scripting languages, *Coda* offers many ways of controlling code flow through conditional control blocks, flow control constructs, etc.

- *IF…ELSEIF…ELSE…ENDIF* – The staple of all programming languages, the IF construct works the same way as Legacy's. It allows one to control the execution of script expressions based on one or more conditions.

  Syntax:

  ```
  IF expressionA [comparison] expressionB
     ; test "expressionA [comparison] expressionB" passed
  ELSEIF expressionB [comparison] expression
     ; test "expressionB [comparison] expressionC" passed
  ELSE
     ; none of the above tests passed
  ENDIF
  ```

  ELSE and ELSEIF clauses are optional. While an IF construct can contain an arbitrary number of ELSEIF clauses, only one ELSE clause is allowed. As with Legacy, *Coda* evaluates the entire conditional expression in IF/ELSEIF statements.

- *WHILE…LOOP* – The WHILE construct is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.

  Syntax:

  ```
  WHILE <expression>
     ; code
     IF <another expression>
        BREAK()
     ELSEIF <yet another expression>
        CONTINUE()
     ENDIF
  LOOP
  ```

  Each WHILE statement should be matched with a corresponding LOOP statement. During execution, the engine evaluates the WHILE expression to a Boolean value. If true, the statements following it will be executed until its LOOP statement is reached, at which point control returns to the top of the loop and the expression is evaluated again. If the expression returns false, execution is returned to the instruction immediately following the LOOP statement.

The *BREAK* and *CONTINUE* commands can be used inside loops to alter the flow of execution – the former causes the loop to exit immediately, while the latter skips the rest of the body of the loop and returns the execution to the top of the loop block for the reevaluation of its condition expression.

- *FOREACH…LOOP* – This unconditional construct is used to iterate over the elements of an array, its workings similar to the WHILE loop.

  Syntax:

  ```
  FOREACH <iterator> <- <expression>
     ; do stuff
  LOOP
  ```

  The expression must evaluate to an array; the iterator can be any variable. At the time of execution, the FOREACH expression is evaluated to get a result of the array data type and the iterator's value is set to the array's first element. Upon reaching the LOOP statement, the iterator is set to the next element in the array and execution returns to the top of the loop. The loop terminates when all of the array's elements have been returned, at which point the iterator is restored to its original value.

- *RETURN* – The RETURN command is used to stop the execution of the calling script, and optionally return a value.

  Syntax:

  ```
  RETURN([expression][, end execution])
  ```

  The command additionally takes a second parameter in background scripts – If '1' is passed, the calling script is permanently removed from the execution queue until it's reinitialized.

  Example:

  ```
  IF (Var1 == 4)
     RETURN()
  ELSE
     RETURN(Var1 * 7 ^ 5)
  ENDIF
  ```

- *CALL* – The CALL command is used to execute external *Coda* scripts as functions. When a script is called as a function, execution passes to the function. The function script continues to execute until its end is reached or a RETURN statement is encountered, at which point execution is returned to the instruction immediately after the function call statement.

  Syntax:

```
CALL("<script name>"[, arguments 1…10])
```

Example:

```
CALL("SwapStrings", Var1, Var2)
Var1 = CALL("ConcatString", "Pig", "gy", "!")
```

The command returns the value passed to the RETURN command, if any, or zero. The first argument must be the name of the function script. Up to 10 arguments can be passed as parameters to the function script. The script name must be the path to the script file relative to the base Coda directory. Backslashes in the path must be escaped, i.e., `CALL("two\\backslashes\\instead\\of\\one\\filename")`. To call a script recursively, use *"SELF"* as the script name.

Coda provides a shorthand notation to call scripts that is syntactically closer to function calls in other programming languages.

Syntax:

```
@<script name>([arguments 1…10])
```

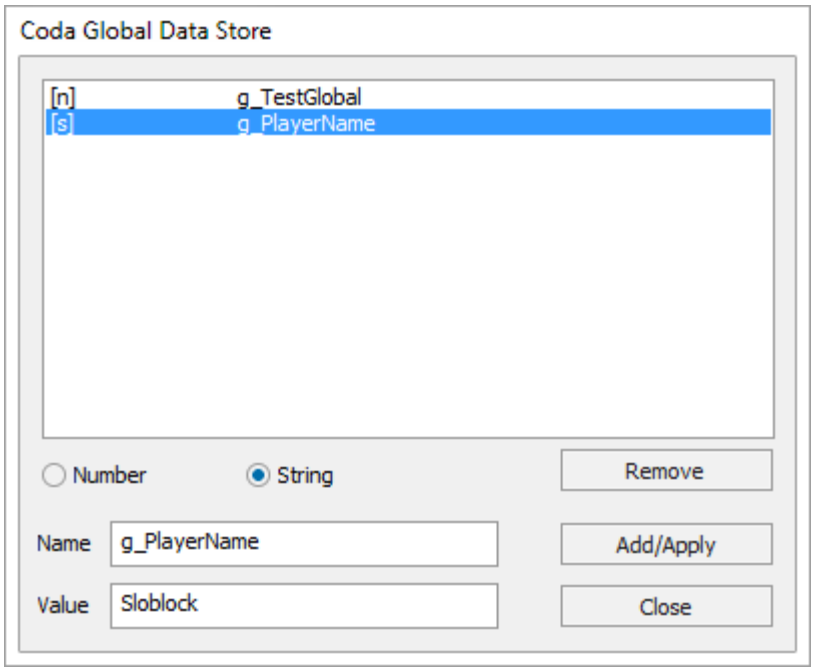
In the above notation, all backslashes in the script name are replaced with a period.

Example:

```
@lib.arrayUtil.arSort(array)       ;-- same as CALL("lib\\arrayUtil\\arSort", array)
```

Global Data Store

In addition to local variables, scripts are allowed to access a pool of global variables. These variables are stored in a location known as the 'global data store'. Global variables need to be explicitly created by the user. The global data store's contents can be managed from the *Coda* menu in the editor's main menu.



Global variables can contain either numeric or string data.

Background Scripts

Coda comes with a special module called the 'Backgrounder' that executes scripts as a background editor process. Any script placed in the Background folder is automatically initialized as a background script at editor start-up and will be executed at regular intervals as determined by the backgrounder's and its own polling interval. Background scripts cannot contain a parameter list. They remain persistent i.e. they preserve their state, for the entirety of a session or until they are reinitialized.

The backgrounding process is performed as a synchronous operation, i.e. there may only be one script executing at any given time. The operation can be suspended (and resumed) from the Coda menu in the editor's main menu.

Executing Scripts and Command Documentation

Regular scripts can be executed from the editor extender's console window by using the *RunCodaScript* console command or by drag-dropping a script file into it.

Syntax:

```
RunCodaScript <script name> <bool:run in background>
```

Scripts invoked in this way cannot have a parameter list.

In addition to the above, the *DumpCodaDocs* console command may be used to quickly open the command documentation for reference.

Syntax:

```
DumpCodaDocs
```

The command generates the documentation and saves it in the editor's root directory under the following name: coda_command_doc.htm

Resource Location

As with all Bethesda Game Studios Editor Extender-related resources, *Coda* scripts are to be saved inside the Data\BGSEE directory.

- Data\BGSEE\Coda – All regular scripts are saved in this folder (or its subfolders, excluding the folder mentioned below).
- Data\BGSEE\Coda\Background – Background scripts are saved in this folder.

Coda scripts are expected to have a file extension of '*coda*'. For instance,
Data\BGSEE\Coda\SampleCodaScript.coda

Appendix

NOTES:

- Arithmetic performed on large numbers can be unreliable due to precision issues. This is an intrinsic limitation of the platform.
- When a script is loaded for the first time, it's compiled to bytecode and cached by the engine. All subsequent executions will reuse the same bytecode. So, changes made to its source code after caching will not be reflected in its execution until the engine recompiles the script.
  The in-memory script cache can be cleared by using the 'Reset Cache' tool in the Coda main menu. This action invalidates and stops all executing scripts - Background scripts will need to be manually reloaded. Scripts executed through the console are always recompiled from disk before execution.

SAMPLE SCRIPTS:

Here are some sample scripts for a variety of tasks.

Strip off Leading Digits from Editor IDs

Display an annoying message in the Console incessantly

Place an object in all Loaded Cells, positioned relative to another Object

Replace one Race with Another