QlikView Expressor Datascript

© QlikTech International AB; May 2013

Contents

1 Product Introduction

QlikView Expressor consists of three components.

1. Desktop: The graphical user interface used to develop and test data integration applications.
2. Date Integration Engine: The command line executable that runs QlikView Expressor data integration applications.
3. Repository: The version control system that integrates with Desktop and the Engine, supporting team development and controlled deployment.

While all three components may be installed onto computers running Windows XP, Windows 7, or Windows Server 2003/2008, the standard approach is to install Desktop onto developer computers running Windows XP or Windows 7 and the Engine and Repository onto computers running Windows Server.[1]

A customer generally deploys Desktop onto many developer computers, configuring them to use a common Repository. This arrangement allows developers to work privately, in what is termed a Standalone Workspace, or within a team, in a Repository Workspace. Once an application's development and testing are complete, a compiled version of the application may be checked out of Repository onto the computer hosting the Data Integration Engine. While many deployments install the Engine and Repository onto separate computers, it is perfectly acceptable to install these two components onto the same computer.

The Engine and Repository components of QlikView Expressor must be licensed and will not run until the license is installed. Desktop is functional without an externally applied license, but does not support the Teradata TDT operator until an appropriate purchased license key has been installed.

In order to work on the examples and exercises presented in this document, it is only necessary to install Expressor Desktop.

This document assumes you have a working understanding of Expressor Desktop and will not provide detailed instructions on creating the artifacts needed to implement the examples and exercises. This document also assumes you understand programming concepts such as defining and invoking a function and using control of flow statements (`if…then…else`; `while…do…end`; etc).

[1] QlikView Expressor is currently not supported on Windows 8 or Windows Server 2012.

2 Installing Expressor Desktop

The executable `QlikViewExpressorDesktopInstaller.exe`, which is approximately 200 MB, is downloaded from the QlikView Web site. When you run this application, it confirms that the required prerequisites (for example, the correct .NET framework) are installed and if necessary downloads and installs any prerequisites that are missing. The process then extracts and runs the actual Desktop installer, `QlikViewExpressorDesktopInstaller.msi`.

During this installation, the only decision you will need to make is to select an installation directory if you do not choose to install under the Program Files, 32 bit operating system, or Program Files(x86), 64 bit operating system, directory.

Exercise: Product Installation

1. Obtain **QlikViewExpressorDesktopInstaller.exe**.
    a. Download from the QlikView Web site.
        i. http://www.qlikview.com/us/explore/experience/expressor-download
2. Run the installer by double-clicking on its entry in Windows Explorer.
    a. Accept the default installation suggestions.

3 Rules Editor

The Aggregate, Join, and Transform operators in the Transformers operator grouping are the operators into which you add business logic such as aggregations or data transformations. You can also use these operators to drop attributes from or add attributes to the record being processed.

In order to specify the business logic to perform, you use the Rules Editor to define an Aggregate, Expression, Function, or Lookup Rule. In a Join or Transform operator, an Expression Rule is equivalent to the right hand side of an assignment statement. That is, you write an expression, which may include nested function calls, that returns a single value. This expression may use any number of the input attributes or Expressor runtime parameters as arguments and its return value is used to initialize a single output attribute.

If your business logic cannot be implemented as an assignment statement, you use a Function Rule. With a Function Rule you are able to use any sort of programming construct, for example, loops or `if <condition>..then..else` blocks, to carry out your data transformations. If necessary, you can always convert an Expression Rule into a Function Rule without losing any of the work you have already done.

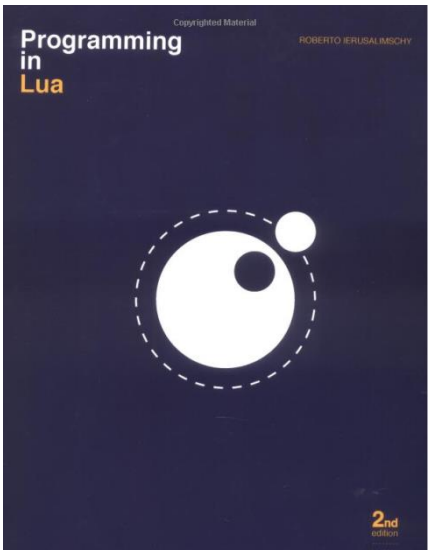
While Expression Rules will most likely support your processing objectives, Function Rules give you the most control over the processing and are therefore significantly more powerful.

The Filter operator, in the Utility operator grouping, also offers a Rules Editor. Code that you include in this operator may only return the values true or false, which then determine how the record will be processed.

Although you are free to simply enter your processing logic, the Rules Editor provides many programming aids such as auto-completion and drop down menus that help ensure there are minimal spelling and logic errors. You can also quickly frame out a complex statement such as a loop by selecting a pre-designed code block. And as your scripting becomes more complex, the find/replace editing features will prove useful. All of these coding aids are located on the Edit tab of the Rules Editor.

In the Aggregate operator, an Aggregate Rule requires no scripting. With these rules, you may only specify one of the incoming attributes as an argument. Before you can select what type of aggregation to perform, you must also associate an output argument with the rule. Once you associate the input and output arguments with the rule, a drop down list is populated with valid aggregations for these argument types. For example, if the input parameter is a numeric type, the output parameter must also be numeric and the possible aggregations will differ depending on whether the output attribute's type is integer or decimal. With an Aggregate Rule, the aggregations presented in the drop down list are your only options. If you want to perform more complex processing, you must use a Function Rule.

4 QlikView Expressor Datascript



Expressor Datascript is derived from version 5.1 of the open source scripting language Lua although Datascript includes functions and data types that are not part of Lua and some functionality available in Lua has been removed.

The book shown on the left documents the version of Lua from which Datascript evolved. This book is available from most book retailers but you may download a PDF version.

Although you will find the Lua documentation interesting to read and will learn a lot from this material, function names in Datascript may be different than equivalent functions in Lua, so your primary reference source for Datascript should be the Expressor product documentation and community site knowledge base.

Using Datascript is not difficult, although the syntax may seem a bit unusual when you first begin studying the language. But it won't be long before you find its simple procedural syntax to be quite friendly.

What makes Datascript easy to use is that you do not need to worry about typing variables or freeing memory. Datascript variables are dynamically typed, which allows you to simply assign a value to a variable. However, once a value has been assigned, the variable assumes the type of the value.

Datascript has only one complex data type, the table. The Datascript table may be indexed using numeric or string values. In fact, both types of indices may be used in a single table. Mastery of tables is essential to becoming an accomplished Datascript developer.

Datascript functions may return multiple values. You will see this behavior with many of the functions that are part of Lua and Datascript and you will find this feature to be especially useful when you are developing your own functions. Your code must capture the multiple return values. Simply assign the function's return to multiple variables.

```
var1,var2 = function_that_returns_2_values()
```

Datascript includes multiple libraries of pre-developed functions. For example, there are many string manipulation functions in the `string` library and extensive mathematical functions in the `math` library. Datascript also includes libraries that contain functions that can be used to read from and write to files, functions that can access operating system variables, and functions used to manipulate tables. Additionally, Datascript adds three new data types to the Lua language – Datetime, Decimal, and Integer

– and eight new function libraries – `bit`, `byte`, `datetime`, `decimal`, `is`, `lookup`, `ustring`, and `utility`.

To call a library function, you must preface the function name with the name of the library. For example, to use the string library function that converts a character string to upper case, you would write the statement

```
var = string.upper("lower case string")
```

and `var` would now contain the character string `LOWER CASE STRING`.

Exceptions to this syntax are the functions in the basic grouping, which are used to access environment variables, load and execute external Datascipt coding, provide output messages, or effect data type conversions. Calling these functions does not require a library name prefix.

Datascript also includes several built-in iterator functions that make traversing tables and character strings easy to accomplish.

Another nice feature of Datascript is that it may also be used to write scripts that can carry out processing (for example, generating a listing of files in a directory) outside of an Expressor application or to coordinate execution of multiple related dataflows. Since the same scripting language is used both within and outside of Expressor, it is easy to test code in isolation and then move it into an Expressor operator or datascript module once you are confident of its accuracy.

Lua includes eight data types: nil, Boolean, number (double precision floating point number), string, userdata, function, thread, and table. Expressor Datascript adds a datetime data type, which is actually a numeric representation of a date as the number of seconds after January 1, 2000 (the beginning of the epoch). Although you now know that datetime values could be manipulated through the standard mathematical operators (+, -, *, /), you should not take this approach; always use the functions in the `datetime` library to manipulate this data type. Datascript also includes an `integer` (`long`) data type and a userdata type that implements a `decimal` number and a corresponding decimal function library.

The `datetime`, `integer`, and `decimal` types must be actively created. Generally, you will create a `datetime` type from a string representation of a date using the `string.datetime` function and `integer` and `decimal` types are derived from numbers using the `tointeger`, `tolong` or `todecimal` functions.

5 Getting Started

Before beginning the detailed study of Expressor Datascript and how it is used in a dataflow, let's review some general programming concepts with which you should already be familiar.

5.1 Arithmetic Operators

Everybody is already familiar with the arithmetic operators addition (+), subtraction (-), unary negation (-), multiplication (*), division (/), exponentiation (^), and modulo (%). You use these operators as you would in a programmable calculator or any mathematical expression. In QlikView Expressor Datascript, these operators may be used with string values that can be converted into numbers. For example,

```
"5" + "6"
```
returns 11

5.2 Relational Operators

The relational operators (==, ~=, <, >, <=, >=) also have the same meanings as in other contexts. These operators always return true or false. The equality operator will always return false if the data types of its operands differ. For example,

```
1 == "1"
```
returns false.

5.3 Logical Operators

The logical operators – conjunction (and) and disjunction (or) – do not function exactly as they would in other programming languages. In other programming or scripting languages, these operators return the Boolean values true or false.

In QlikView Expressor Datascript, the logical operators do not necessarily return Boolean values, but rather the value of one of their inputs.

- The conjunction operator returns its first input if that input value is, or evaluates to, false or nil, otherwise it returns the value of its second input.
- The disjunction operator returns the value of its first input if that input value is not false or nil, otherwise it returns the value of its second input.
- A Boolean value is returned only if the appropriate input is itself a Boolean value.
- The conjunction operator has higher precedence than the disjunction operator and both of these operators have lower precedence than the relational and mathematical operators.
- The negation (not) operator is more typical and always returns true or false.
- Only the values false and nil are interpreted as false.
- The values zero, one, and the empty string are all interpreted as non-nil, and therefore true, values.

The last two bullets are very important and bear repeating. In Expressor Datascript, only the values false and nil are interpreted as false. This means all values including empty strings or the numbers zero and one are true.

```
val = (5==6) or "not equal"
```

The variable `val` is initialized to the string 'not equal'. The conditional test returns false, so the disjunction operator evaluates and returns its second argument.

```
val = 5 and (false or 6)
```

The variable `val` is initialized to the number 6. The first argument to the conjunction operator is true (it is a non-nil and non-false value) so the second argument is evaluated and returned. The second argument uses the disjunction operator; its first argument evaluates to false so it evaluates and returns the second argument.

5.4 Control of Flow Statements

Expressor Datascript includes typical control of flow statements.

5.4.1 if <condition>..then..[elseif <condition>..then]..else

The conventional `if <condition>..then..else` block is supported. Note that the keyword `then` must follow the conditional test. If desired, nested `elseif <condition>..then` blocks may be included. Execution runs through this control of flow statement only once.

5.4.2 while <condition>..do..end

The condition is tested and if it evaluates to true, the block is executed. Note that the keyword `do` must follow the conditional test. Processing then re-evaluates the condition and re-executes the block if the condition is still true. This process repeats until the condition evaluates to false. If the condition returns false in its initial evaluation, the block of code is never executed.

5.4.3 repeat..until <condition>

The block of code is evaluated before the condition is tested. If the condition evaluates to true, the block is re-executed and the condition re-evaluated. This process repeats until the condition evaluates to false. This block of code will be executed at least one time.

5.4.4 for <looping directives> .. do..end

The looping directives set a starting number, an ending number, and an optional incrementing value (default is 1; may be negative). The block of code will be executed for each value between the starting and ending numbers adjusting the starting value by the incrementing value with each execution. Looping continues until the starting number equals or exceeds the ending number. If the stopping condition exists when the loop is first entered, the block of code is never executed.

5.4.5 break and return

The `break` and `return` statements allow exit from a block of code. The `break` statement is used to exit a loop – `while`, `repeat`, `for` – and cannot be used outside of one of these control of flow statements. Execution restarts at the first statement immediately after the loop's closing `end` statement.

The `return` statement emits a result from a function.

6 Mastering Expressor Datascript

In many situations, the graphical editors that are part of QlikView Expressor Desktop will provide sufficient support for you to develop your coding while creating a dataflow. But in other situations, you may need to write more extensive processing logic that needs to be developed and tested outside the scope of a running dataflow. While you could write this code using Desktop's Rules Editor, you may find it helpful to work outside of this editor for this type of work. Additionally, external scripts – called Datascript Modules – that you may want to develop are written in a dedicated editor and cannot be debugged from within the editors used to configure the programmable operators. QlikView Expressor provides a command line utility – **datascript** – that you can use to develop and test your code.

To use this utility, you must open a command window from the **Start – All Programs – QlikView – expressor3 – expressor command prompt** menu item. Opening a command window using other approaches will not properly set the environment to use **datascript**.

The **datascript** utility may be run interactively, in which case each line of a script being developed or tested is entered individually into the command window, or a script may be contained in a file and executed as a single entity.

To start the utility in interactive mode, open a command window and enter **datascript** followed by **Enter**. Then type each line of code to be tested. Pressing **Enter** before completing a statement is permitted. The interpreter will wait until the statement is complete and followed by **Enter** before processing. Use **Ctrl-C** to exit the interactive session.

Alternatively, you can place multiple lines of code into a text file and execute within a command window by entering **datascript path\file_name** followed by **Enter**.

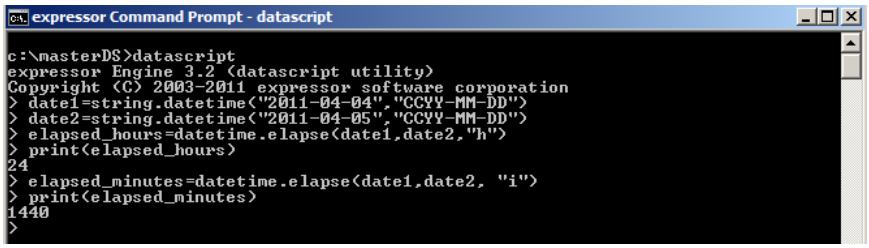
- It is possible to include arguments in this command, which your code can then pick up and use.
- The arguments are provided as a zero indexed array (actually as a Datascript table, but at this point you aren't ready to work with table notation) named `arg`.
    - `arg[0]` is the name of the script file being run
    - `arg[1]` to `arg[n]` are the arguments you are passing into the script
        - All arguments are retrieved as string types, so your code may need to perform a type conversion before using.
        - There is no need to enclose string values within quotation marks unless they include embedded space characters.

Running the **datascript** utility in a command window is the best way to master each of the QlikView Expressor Datascript functions and learn to use the various control structures.

Be certain to open a command window using the **Start – All Programs – QlikView –expressor3 – expressor command prompt** menu item, then change directories into a directory you will use to hold

any script files you may be developing. Then issue the **datascript** command to enter interactive mode or use this utility to execute the code in your script file.

To experiment with a function, take it step-by-step until you feel confident enough to build up a single statement or block of code that accomplishes the entire objective. The **datascript** utility also includes a `print` function that you can use to display the value in any variable or the return from a function call, as shown in the following figure. The `print` function accepts a comma separated list of values to display.



Even when your coding skills have improved and you feel confident enough to code within an operator's Rules Editor it's useful to use **datascript** to test your syntax and logic.

In doing the exercises in this course, it is recommended that you create a working directory, for example, `C:\mywork`, in which you will store text files containing your exercise scripts. With this approach you can easily modify and rerun a script until you have mastered the concept. If you want to insert a comment in a script, preface the line with two dash characters.

```
-- this is a comment that extends to the end of the line
```

A section at the end of this document includes descriptions for the function libraries most useful in completing the exercises. The product documentation includes descriptions of all the function libraries.

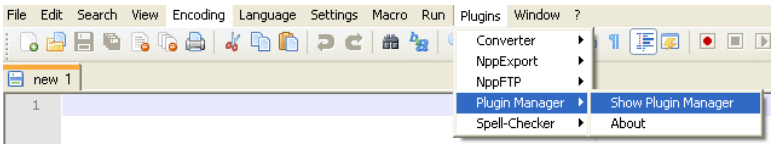
6.1 Scripting Approaches

To try a few Datascript functions, such as the example shown in the screen shot above, using the **datascript** utility in interactive mode is sufficient. But to develop a block of code or several interacting functions, the scripting should be done in a text file so that edits and testing can proceed more rapidly. What's a good approach for writing these script files?

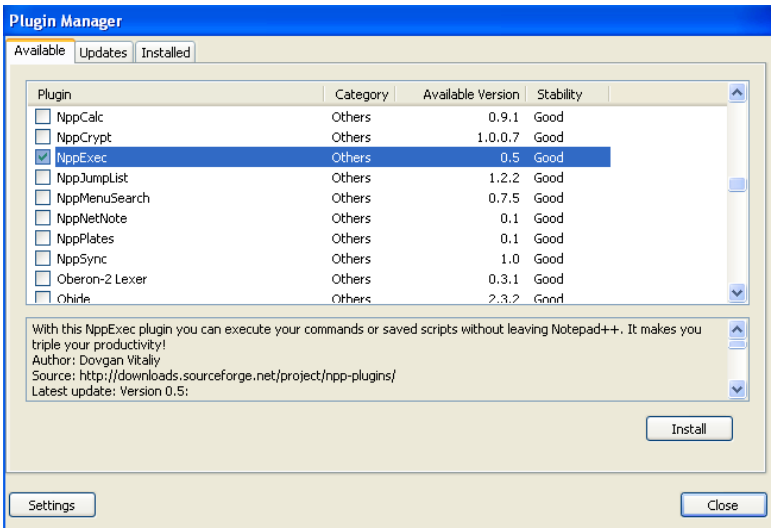
Any text editor will do, so Notepad or WordPad are valid choices, but neither of these tools provides any sort of coding support such as text completion or text formatting (colored font, indentation) to help with the scripting effort.

Another approach is to use Notepad++ for both scripting and testing code. This editor is Lua-aware so it will provide syntax support during scripting and, by installing the NppExec plugin, scripts can be run without the need to open a command window.

- Start Notepad++ and click on the **Plugins > Plugin Manager > Show Plugin Manager** menu item.



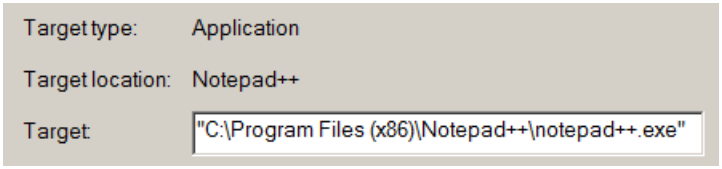
- Select the Available tab and in the listing of plugins, scroll to, select, and install the **NppExec** plugin.



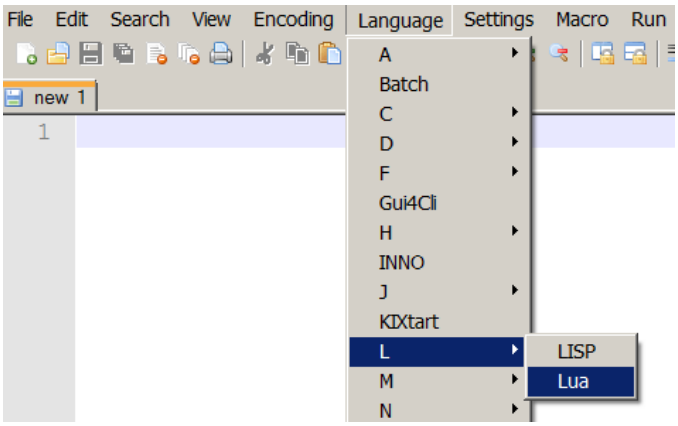
- Shut down Notepad++.
- You only need to install this plugin one time, not each time you want to use Notepad++ to work with Expressor Datascript.

To use Notepad++ to develop and test Datascript, it must be started from an Expressor command window so it picks up the Expressor Datascript libraries.

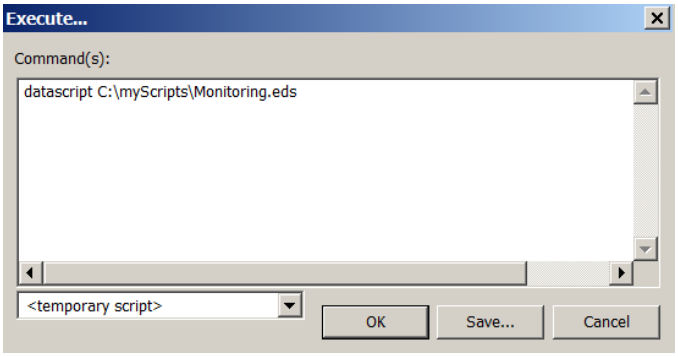
- From the Windows Start menu, drill down to the entry for Notepad++ and open its Properties sheet.
  - Copy the application start command from the Target text box control.



- Now open an Expressor command window using the **Start – All Programs – QlikView –expressor3 – expressor command prompt** menu item.
  - Paste the application start command into the window and press **Enter**.
  - Notepad++ will start with access to the Expressor Datascript libraries.
- Set the language type to Lua.



- Notepad++ will provide support for coding in Datascript.
- Use the Notepad++ editor to create your script.
    - Save the script file to a convenient file system location.
- To run the script, select the **Plugins > NppExec > Execute** menu item (or press F6), and in the Execute… window, enter the **datascript** command with or without additional arguments: **datascript path\file_name**



    - Note that the script being run does not need to be open in the editor.
    - Print statements included in the code as well as errors raised on execution will display in the Notepad++ Console panel.

7 IS Library

Since the type a variable represents is set dynamically when the variable is initialized, it is necessary to have a function that can determine the type of a variable. In the group of basic functions, the `type` function serves this purpose. In addition, the `is` function library includes functions that test the type of a variable returning true or false.

To initialize a variable with a value, write a simple assignment statement such as

```
number_var = 25
string_var = "hello"
```

Note that uninitialized variables are nil. That is, referring to a variable that has not been initialized will return the value nil.

```
print(var)
nil
var = 10
print(var)
10
```

7.1 The is Library Functions – Data Type Exercises

Before beginning the following exercises, review the `is` library functions in the Appendix to this document.

1. Set some variables to the following values.
    a. Nil
        i. Remember, any uninitialized variable will evaluate to nil.
    b. “”
    c. “ “
    d. 25
    e. true
    f. “expressor software”
2. Test each variable with the `type` function.

nil	“”	“ “	25	true	“expressor software”

Are you confused with respect to the type returned for 25? Why? Perhaps filling in the following table will provide an answer.

3. Test each value with the various `is` functions and summarize your findings.

	is.blank	is.decimal	is.datetime	is.empty	is.future	is.integer	is.null	is.number	is.past
nil									
“”									
“ “									
25									
true									
“expressor software”									

Still confused? Remember, the `integer` and `decimal` types must be actively created. Review the `todecimal`, `tointeger`, `tolong`, `tonumber`, and `tostring` functions in the basic grouping.

4. Use `is.pattern` to extract “Expressor” from the value “Expressor Datascript”.

8 OS Library

This library contains functions that your code can use to interact with the Windows operating system. For example, the `os.getenv` function can be used to retrieve the value of an environment variable and the `os.remove` and `os.rename` functions can be used to delete or rename files.

While this library includes functions that return date and time values, your code should not use these functions. Whenever your code needs to manipulate data or time values, you should use the functions in the `datetime` library.

Probably the most important function in the `os` library is `os.execute`. This function can be used within a script to run an Expressor dataflow. Simply provide this function with a string argument that contains the `etask` command you want to run.

8.1 The os Library Functions

Before beginning the following exercises, review the `os` library functions in the Appendix to this document.

1. Retrieve and display the values of the following environment variables set on your computer:
    a. USERNAME
    b. USERPROFILE
    c. PATH
2. Create a file in the same file system location in which you are running the **datascript** utility.
    a. The name of the file doesn't matter and it doesn't matter what's in the file, so just enter a few lines of text.
    b. Use an `os` library function to change the name of this file.
    c. Open an Expressor command window to another file system location (not the same directory that contains the file).
        i. Change the name of the file again.
            1. Did you need to change the argument to the function call?
            2. How can you determine that the function call succeeded?
                a. Looking for the file isn't the answer.

9 IO Library

The `io` library includes two collections of functions for reading from and writing to files. If your code only needs to read from or write to a single file, then you could use the simple IO functions, whereas code that requires simultaneous access to multiple files for reading or writing must use the complete IO functions. Regardless of which set of functions you are using, you will be able to read files either row by row or as a single large string. Writing to files is always line by line. When you read a file line by line, the newline character at the end of each line is dropped; when you write to a file, your code must write the newline character at the end of each line.

The primary difference between the simple IO and complete IO functions is the way in which you open a file. With the simple IO approach, there are separate functions for opening a file for reading or writing whereas the complete IO approach uses a single function for which an argument indicates the operation that will be performed on the file. The complete IO approach also allows a file to be opened in a separate process. Since the complete IO approach is more functional, this approach will be emphasized in this material.

Think about the differences between the function `read`, which returns single lines or the entire file, and the function `lines`, which iterates through the file returning one line at a time.

```
file_handle = io.open("somefile", "r")
line = file_handle:read("*l") -- that's a lower case L, not a 1
```

Versus

```
file_handle = io.open("somefile", "r")
for line in file_handle:lines() do …… end
```

9.1 The io Library Functions

Before beginning the following exercises, review the `io` library functions in the Appendix to this document.

1. In some convenient file system location, create a simple text file in which each line holds a single word or person's name. Add 5 or so lines to the file.
    a. Read and display the contents of the file.
2. Read the file's contents and write to another file.
    a. Did your code read a line then write the line?
    b. As soon as you learn a bit more about Datascript, iterative functions, and control of flow structures you will be able to use the `lines` function or use the `read` function to read the file in one large 'gulp' before writing line by line.

10 Expressor Datascript Tables

Expressor Datascript tables are extremely powerful and useful. Fortunately, basic uses of the table are easy to master and the more complex usages are not that much more involved.

In Expressor Datascript, tables are the only complex data structure. Tables have no fixed size, can be indexed using numbers, strings, or a combination of both numbers and strings, and can change size as needed. Table entries evaluate to nil when they are not initialized.

Let's consider the basic numeric indexed table. In this usage, index values start, by default, at one, although you can specify zero or negative numbers as the index value. If you want to accept the default indexing scheme, you can declare and initialize the table in a single statement.

```
myTable = {val1, val2, val3, ... }
```

If `val` is a numeric, you simply enter it. If a table value is a string, it must be enclosed in quotation marks (`"`). To retrieve a value, you specify the index of the value you want.

`myTable[1]` returns `val1`, while `myTable[2]` and `myTable[3]` return `val2` and `val3`

If you want to set the index value, you must explicitly provide it.

```
myTable = {[-1]=val1, [-2]=val2, val3, ... }
```

In this case, `val1` is associated with the index -1, `val2` with the index -2, and `val3` with the index 1. You are free to intermix specified and default indices.

```
myTable = {[-1]=1, [-2]=2, 3, [-3]=4, 5}
```

Although you can use negative numbers as indices, this is not a common practice as it makes retrieving values from the table more difficult.

Table indices are not limited to numbers, strings are acceptable too. But there are a few rules you need to observe.

- If the index starts with an alphabetic character and includes only alphanumeric characters, it may be entered without surrounding quotation marks (`first` and `s2` in the following example).
  ```
  myTable = {first=val1, s2=val2}
  ```
  - To retrieve the value you can use one of two notations:
    `myTable["first"]` or `myTable.first`
- If the index starts with a numeric or non-alphabetic character, or includes a non-alphanumeric character, it must be entered with surrounding quotation marks within a set of square braces.
  ```
  myTable = {["1"]=val1, ["2!"]=val2}
  ```
  - To retrieve the value you must quote the index value:
    `myTable["1"]` or `myTable["2!"]`

Of course, you can always declare the table first and then add the elements.

```
myTable = { }
myTable[1] = val1
myTable["2!"] = val2
```

Note that with this approach, you must always provide the index. Note also, that you are free to mix numeric and string indices in the same table.

A table may hold complex types, for example other tables.

```
Table1 = {1,2}
Table2 = {first=Table1, second = {3,4}}
```

In `Table2`, the index `first` references `Table1` while the index `second` references an unnamed table. To retrieve one of the numeric values, you must specify two indices.

`Table2.first[1]`, which returns the value 1
`Table2.second[2]`, which returns the value 4

Or consider

```
Table2 = {Table1}
```

where 1 is the index in Table2 that corresponds to Table1. To retrieve values, use a two-dimensional index:

`Table2[1][2]`, which returns the value 2

Another way to quickly initialize a table is to assign the return values from a function that generates multiple return values. For example, the `string.match` function can be used to parse an IP address into its constituent parts and each part used to initialize an element of the table. Note how the function call is enclosed in curly braces `{string.match(…)}`, which creates a numerically indexed table from the function's multiple returns.

```
Table3 = {string.match("127.0.0.1","(%d+)%.(%d+)%.(%d+)%.(%d+)")
```

The pattern `(%d+)%.` looks for a sequence of one or more digits followed by a dot and the parentheses capture the digits as a return value. `Table3[1]` now contains 127, `Table3[2]` contains 0, `Table3[3]` contains 0, and `Table3[4]` contains 1, and you can easily retrieve these values from the table.

And perhaps one of the most interesting uses of a table is to hold functions.

```
myTable = { }
function myTable.larger(a,b)
  return (a>b) and a or b
end

myTable.larger(4,5) returns 5
myTable.larger(4,3) returns 4
```

Here, the table `myTable` uses the index `larger` to locate the code that returns the greater of two numbers. This is a nice technique to use to build up your own named library of functions. In fact, this is exactly how the functions in the `string`, `datetime`, or other libraries have been defined.

So, how do you retrieve values from a table? As you saw above, you can always provide the index of the element you want to retrieve. But what if you don't know the index for your desired entry, is there a way to work through a table, retrieving values? The answer is "Yes"!

QlikView Expressor Datascript includes two iterator functions that are designed to walk through a table returning each index and its associated value.

With a numerically indexed table that uses the default indexing scheme, this iterator function is named `ipairs`.

```
for index, value in ipairs(myTable) do
  -- manipulate index and value in some way
end
```

This code block will retrieve each index and value from the table, starting at index 1, and continue until all sequentially indexed elements have been retrieved. Values associated with zero or a negative numeric index or with a string index will not be retrieved. The iterator will stop retrieving values as soon as an out of order numeric or string index is retrieved.

With a table that uses string and/or numeric indices, even negative numeric indices, the iterator function `pairs` will walk through the table.

```
for index, value in pairs(myTable) do
  -- manipulate index and value in some way
end
```

You will find many uses for tables when working with QlikView Expressor Datascript. In fact, any time you need to create a collection or want to implement an efficient way to carry out a series of comparisons, such as nested `if..then..else` statements, or perform a lookup, think table.

In order to optimize memory usage, tables have their own memory management rules. Normally when you assign a value to a variable, the variable holds a private copy of the value.

```
var1 = "hello"
var2 = var1       --both variables hold the same value
var2 = "goodbye"  --var1 holds "hello"; var2 holds "goodbye"
```

This is not true with tables.

```
var1 = {"first","second"}
var2 = var1
var2[1] = "third" --var1[1] and var2[1] hold "third"
```

That is, the assignment statement `var2 = var1` does not copy the values within `var1` into `var2`; rather the two variables point to the same location in memory and any changes to a value in `var1` will be reflected in `var2` and *vice versa*. If you want to copy a value, you must explicitly reference the element.

```
var1 = {"first","second"}
var2 = {var1[1], var1[2]}
var2[1] = "third"   --var2[1] holds "third"; var1[1] holds "first"
```

10.1 Table Exercises – The table Library Functions

Before beginning the following exercises, review the `table` library functions in the Appendix to this document.

1. Initialize a numerically indexed table with the three letter airport abbreviations; five or six will be sufficient.
    a. Using the `ipairs` iterator function, print out the contents of the table, one airport abbreviation on each line.
        i. Print out each index value and its associated airport abbreviation, one pairing per line.
            1. The print function may take multiple comma separated arguments.
    b. Now, print a single line with all the airport abbreviations where the individual entries are separated by a comma, pipe character, or tab.
    c. Add another airport abbreviation to the table.
        i. There are two ways to do this.
        ii. Investigate both ways.
    d. Sort the contents of the table in alphabetical order and display as in 1.b.
    e. Finally, sort the contents of the table in reverse alphabetical order and display as in 1.b.
2. You have an application in which three letter abbreviations are used for airport identifiers (assume that there are no spelling errors in the incoming data, but the case of the characters is not necessarily consistent).

a. You need to convert each airport abbreviation into the full name; for example, BOS needs to be converted to Boston.
    i. Develop a routine that will look up each airport's full name when supplied with the airport abbreviation.
        1. Again, five or six airport name, abbreviation pairings would be sufficient.
    ii. Investigate what happens when a meaningless abbreviation is submitted to your routine.
        1. How can you exploit this situation?
b. Revise your routine so that the input could be either the airport abbreviation or name but the return is always the airport name.

11 String Library

Probably the functions that you will use the most often are included in the `string` library. If your application data uses Unicode characters, you will need to use the `ustring` library, which includes a collection of identical functions to the `string` library. This document will only discuss the string library, but you now know that Datascript can just as easily work with Unicode data.

11.1 Function Details

The string library includes 30 functions, most of which are straight-forward to master. The five functions that have a bit of a learning curve are `string.datetime`, `string.find`, `string.iterate`, `string.match`, and `string.substring`.

11.1.1 string.datetime

This function converts a string representation of a date, time or date and time into a `datetime` type. Except when you read date or time types from a database table or Excel worksheet, your code will need to use this function to create `datetime` values. So, it is very important that you master this function.

This function requires two arguments, although the underlying implementation will provide a default value for the second argument. The first argument is the string representation of a date or time while the second argument describes the string representation.

For example, "January 1, 2013" is a string representation of a date, but so are "1 JAN 2013" and "01/01/2013" and many other possible representations. And "12:15" and "01/01/2013 12:15:30" are representations that include time information. The `string.datetime` function can convert any of these representations into a `datetime` type, but to do so it needs your help. The second argument to this function must describe the format of the representation.

String Representation	Format
January 1, 2013	MMMM D*, CCYY
1 JAN 2013	D* MMM CCYY
01/01/2013	M*/D*/CCYY
12:15	HH24:MI
01/01/2013 12:15:30	M*/D*CCYY HH23:MI:SS

When calling the `string.datetime` function (or the `datetime.string` function), the format **ALWAYS** refers to the string representation. For `string.datetime`, the format describes the incoming value (for `datetime.string`, the format describes the outgoing value). Within Expressor, `datetime` values are represented in a default format (CCYY-MM-DD HH24:MI:SS). If you omit the format argument, it is assumed that the string representation uses/will use the default format. If only a time value is represented (for example, 12:15), within Expressor the corresponding `datetime` value will be 2000-01-01 12:15, that is, the date is set to the beginning of the epoch.

11.1.2 string.find

This function examines a character string for a specified character pattern, return the positions at which the pattern begins and ends. To fully utilize the power of this function, you will need to master the Expressor Datascript/Lua pattern matching syntax, but a simple example will give you an idea how this function works.

```
character_sequence = "Expressor Datascript"
beginning_position,ending_position =
   string.find(character_sequence,"Data")
-- beginning_postion will hold 11
-- ending_position will hold 14
```

If you want the function to also return the character string that satisfies the pattern, enclose the pattern in parentheses. The returned character string is referred to as a capture.

```
character_sequence = "Expressor Datascript"
beginning_position,ending_position,character_strng =
   string.find(character_sequence,"(Data)")
-- beginning_postion will hold 11
-- ending_position will hold 14
-- character_string will hold "Data"
```

11.1.3 string.iterate

This function walks across a character string, returning character sequences that match a specified pattern. To fully utilize the power of this function, you will need to master the Expressor Datascript/Lua pattern matching syntax, but a simple example will give you an idea how this function works.

```
character_sequence = "Expressor Datascript"
characters = {}  -- create an empty table
for character in string.iterate(character_sequence,"(s)") do
  -- add each extracted character to the table
  characters[#characters+1] = character
end
-- examine each extracted character
for k,v in ipairs(characters) do
  print(k,v)
end
```

The `string.iterate` function extracts each character that is a lower case S, placing it into the variable `character`. Each returned value is added to the numerically indexed table `characters` then the `ipairs` iterator function displays the contents of the table (three entries containing's').

Stop and think!

Go back to the `io` library and the option to read an entire file in one 'gulp.' You could use the `string.iterate` function to parse the file's content into individual lines by looking for character strings that DO NOT include control characters (that is, all characters up to the newline indicator).

11.1.4 string.match

This function is similar to `string.find` except that it returns only the character string that satisfies the pattern. If no match is found, the function returns nil.

```
character_sequence = "Expressor Datascript"
matching_pattern =
   string.match(character_sequence,"Data")
-- matching_pattern will hold "Data"
```

An optional third argument can be used to specify a starting position within the source string.

```
character_sequence = "Expressor Datascript"
matching_pattern =
   string.match(character_sequence,"Data",10)
-- begins searching at character 10, the space between Expressor
--   and Datascript
-- matching_pattern will hold "Data"
```

11.1.5 string.substring

You will find many uses for the `string.substring` function. It's not hard to use, but it is different from the substring implementations in other programming and scripting languages. This function takes three arguments, although the implementation will provide a default values for the second and third arguments.

The first argument is the string on which you want to act. The second argument is the position of the character on which to begin the substring action and the third argument is the position of the character on which to end the substring action. Many other substring implementation use the third argument to specify the number of characters to substring.

If you do not provide a value for the second argument, Datascript assumes position 1. If you provide a negative value, the starting point is relative to the last character of the string. For example, if the string is 'Expressor' and the second argument is -7, the substring will begin with the 'p' character. If you do not provide a value for the third argument, Datascript assumes the position of the last character.

11.1.6 Datascript/Lua Pattern Matching Syntax

To use the `string.find`, `string.iterate`, and `string.match` functions you need to be comfortable with the Datascript/Lua pattern matching syntax described in the Appendix to this document.

When specifying a pattern constraint for an attribute in a composite type or an atomic type, you use a different pattern matching syntax based on standard regular expressions.

Although these two pattern matching syntaxes share similarities, there are some significant differences between them, for example, the escape character, the interpretation of how parentheses are used, and the interpretation of how curly braces are used. Also, the standard regular expression based syntax has more ways to specify pattern elements.

You will need to master both syntaxes, but this document only details the Datascript/Lua pattern matching syntax. See the pattern matching related information in the Appendix to this document.

11.2 The string Library Functions

Before beginning the following exercises, review the `string` library functions.

1. Using the `string.find` and/or `string.match` functions, extract the top-level domain from an email address. Be sure to consider the fact that there may be periods before the @ character in an email address.
    a. Your solution should be able to handle top-level domains of 2 to 4 characters.
    b. How would you extract a multi-part domain such as `.co.in`.
        i. Will this pattern also work for example 1.a?
2. Suppose you have a variable initialized to a string value that includes characters that you want to remove. For example:

    ```
    123 Main Street 'Apt 25'
    ```

    a. Investigate various ways of removing the single quote characters from the string.
        i. There are three different functions you could use.
        ii. Investigate the differences between these functions.
3. Write a routine that determines how many times the characters "ss" appear in the word Mississippi.
4. Go back to the `io` library exercises and
    a. Use a file handle to read the entire content of a file in a single 'gulp.'
    b. Then use the `string.iterate` function to write each line individually to another file.
        i. The pattern you should use to separately extract each line is "`(%C+)`".
            1. The quotation marks are required.
            2. The parentheses indicate that the character string satisfying the enclosed pattern should be returned.
            3. %C+ indicates one or more non-control characters.
                a. That is, the line's content up to the newline.

12 Datetime Library

The functions in the `datetime` library are extremely useful for performing calculations with date and/or time values.

12.1 Function Details

The functions that will be used the most are `datetime.adjust`, `datetime.elapse`, `datetime.moment`, and `datetime.string`. You might tend to use the `datetime.timestamp` function in your code to obtain the current date. While you can certainly do this, it's not the best approach as each time the function is called it could return a slightly different value as the time does change during execution of a dataflow. If you are using this value to timestamp records as they are being processed, records will have slightly different timestamp values and it may be difficult to subsequently select the entire group of records, for example, through a SQL query with a WHERE clause that refers to the timestamp value. The better approach is to use the value in the runtime parameter `dataflow.start`, which is the time at which execution of the dataflow began. This value will not change as the dataflow runs and you will be entering the same timestamp value to every record.

12.1.1 datetime.adjust

Use this function to change a date by a predetermined interval, for example, a year, day, hour. This function requires three arguments, although the third argument is optional. Provide the `datetime` value you want to adjust, the interval (as a number), and the optional format (year, day, etc) as the arguments. If the format value isn't provided, the function assumes seconds. The third argument is a string and must be enclosed in quotation marks. An adjusted `datetime` value is returned.

12.1.2 datetime.elapse

Use this function to determine the interval between two `datetime` values. This function requires three arguments, although the third argument is optional. Provide the two `datetime` values as the first and second arguments and the format (year, day, etc) in which you want the interval expressed as the third argument. If the third argument is omitted, the function assumes seconds. The third argument is a string and must be enclosed in quotation marks. The function returns the interval between the two values as a number.

12.1.3 datetime.moment

Use this function to parse a `datetime` value into its constituent parts, that is, month, day, year, century, hour, second, etc. This function requires two arguments, although the second argument is optional. Provide the `datetime` value you want to parse as the first argument and the part you want to retrieve as the second argument. If the second argument is omitted, the function assumes seconds. The second argument is a string and must be enclosed in quotation marks. The function returns the desired part. Note that this function returns the day of the week as a number (0 is Sunday, 6 is Saturday) rather than the name of the day and the month as a number rather than the name of the

month. To retrieve the actual day or month name, use the `datetime.string` function with a format that specifies only the day or month name.

12.1.4 datetime.string

This function converts a `datetime` value into a string representation. This function requires two arguments, although the second argument is optional. Provide a string representation of a datetime as the first argument and the format of the datetime representation as the second argument. If the second argument is omitted, the function assumes `CCYY-MM-DD HH24:MI:SS`. The second argument is a string and must be enclosed in quotation marks.

Note that the `string.datetime` and `datetime.string` functions use the same formatting syntax to describe the string representation of a datetime value.

12.2 The datetime Library Functions

Before beginning the following exercises, review the `datetime` library functions in the Appendix to this document.

1. Using your birthday and today's date, determine how many days old you are.
2. How many minutes and seconds are there in an eight hour workday?
3. Determine the day of the week (Sunday – Saturday, not 0 to 6) for your birthday in the year you were born and next year.
    a. There are two ways to do this.
4. What day of the week (Sunday – Saturday) will it be 30 days from today.
    a. Likewise, there are two ways to do this.

13 Utility Library

The utility library includes a group of functions that you can use to store values in and retrieve values from the persistent store that is integrated into Expressor. This persistent store is actually a lightweight relational database management system that is also used to store and manage lookup tables. You can use this persistent store to pass information between dataflows or to pass data into a dataflow.

For example, if you want to save the last customer account number so that the next time you run the dataflow that generates new account numbers the account number sequence can begin with the proper value, you would use this persistent store.

13.1 Function Details

There is a separate function to store values of each Expressor data type. Similarly, there is a separate function to retrieve values of each type. The functions that store values require two arguments: a string that is the name under which the value will be stored; and the value. The functions that retrieve values require one argument: a string that is the name of the value to retrieve.

13.2 The utility Library Functions

Before beginning the following exercises, review the `utility` library functions in the Appendix to this document.

1. Use the `utility.store_string` function to store an absolute path entry.
    a. This path should include a nested directory.
        i. For example, `C:\myfiles\data`.
    b. Remember, the `\` character is a special character needs to be escaped.
        i. The / character is not a special character.
2. Use the `utility.retrieve_string` function to confirm that the stored value is an interpretable path.

14 Basic Function Grouping

In addition to the type conversion functions (`todecimal`, etc), this group includes some very useful functions: `decision`, `decode`, `ipairs`, `pairs`, `pcall`, and `select`.

14.1 Function Details

In an earlier section on Datascript tables, you saw the utility of the `ipairs` and `pairs` functions when retrieving the contents of a table. The `decision` and `decode` functions provide ways to mimic the processing of a CASE statement or IF..THEN..ELSE or IF..THEN..ELSEIF..ELSE statement in an expression rule. The `select` function allows you to limit the return values from a function that returns multiple values. And the `pcall` function is used to call a function in protected mode, which prevents a fault in the protected function from raising an error in your invoking code.

14.1.1 decision

When you use the `decision` function, you do not need to explicitly code a return statement. Rather, the function will return a value that you specify. Consequently, you can use this function to implement the equivalent of a complex block of code while adhering to the requirements of an expression rule.

The `decision` function sets up a series of logical comparisons, each associated with a return value. The function evaluates each comparison in sequence and returns the return value for the first comparison that evaluates to true. If none of the comparisons return true, the function can emit a default return value.

```
decision(comparison_1,return_1,…,comparison_n,return_n,default_return)
```

If comparison_1 (which might be val1==val2 or val1<val2) evaluates to true then the function emits return_1, if not, it evaluates comparison_2, and if true emits return_2. This continues through comparison_n and if none of the evaluations return true, the default_return value is emitted.

14.1.2 decode

When you use the `decode` function, you do not need to explicitly code a return statement. Rather, the function will return a value that you specify. Consequently, you can use this function to implement the equivalent of a complex block of code while adhering to the requirements of an expression rule.

The `decode` function sets up a series of logical comparisons, each associated with a return value. The function evaluates each comparison in sequence and returns the return value for the first comparison that evaluates to true. If none of the comparisons return true, the function can emit a default return value.

```
decode(value,comparison_1,return_1,…,comparison_n,return_n,default_return)
```

If value==comparison_1 then the function emits return_1, if not, it evaluates value against comparison_2, and if true emits return_2. This continues through comparison_n and if none of the evaluations return true, the default_return value is emitted.

14.1.3 select

The `select` function requires two arguments, the number of the first of the multiple return values you want to receive and the arguments or expression to evaluate.

```
function multi_return()
  return 1,2,3,4 -- returns four values
end

a,b = select(3,multi_return())
```

The variable a will contain the value 3 and the variable b will contain 4.

14.1.4 pcall

This function allows you to isolate/protect a function call so that any errors the function might raise are trapped and processing is not terminated. If execution of the protected function succeeds, `pcall` returns true and the function result; if execution raises an error, `pcall` returns false and an error message. The arguments to `pcall` are the function name and a comma separated listing of the function's arguments. For example, suppose you want to convert a string datetime representation into a `datetime` value but there are several possible formats. If you invoke `string.datetime` with the wrong format, it will raise an error that stops further processing. But encapsulating this call in `pcall` will allow the processing to continue.

```
date_value = "January 1, 2013"
datetime_value = string.datetime(date_value,"…")
```

Unless the format (…) is "`MMMM D*, CCYY`", the invocation of `string.datetime` will raise an error. But by using `pcall`

```
success,datetime_value = pcall(string.datetime,date_value,"…")
```

the error is isolated from the main processing thread. If `success` is true, the conversion succeeded; if it is false, conversion failed.

14.2 The basic Grouping Functions

1. You've been given two lists of children's names.
    a. One list contains typical boys' names.
    b. The second list contains typical girls' names.
    c. Using the `decode` function, write code that determines if a name is masculine or feminine; if neither, return "sex unknown."
    d. Now do the same things using the `decision` function.

    You do not need to keep the two listings separate; it's OK to combine them into a single listing if that is appropriate for your implementation.

2. You have an assignment to create an application that processes records that contain date values. Despite assurances that all date values will be entered as MM/DD/CCYY (that is, always two character month, two character day, and four character century/year), that isn't so. Some records have dates entered in different formats, for example, single character months or days, or the century/year before the month and day.
    a. In your application, you need to convert these string date representations into `datetime` values, but you know you cannot blindly accept that all entries will have the same format.
    b. Write a routine that is capable of determining what format is being used to that the `string.datetime` function can be successfully invoked for all values.
        i. Your approach should be robust enough so that a new format can be easily incorporated into the processing logic.

15 Datascript Modules

One of the competitive advantages of QlikView Expressor is extensive support for writing reusable code that you can incorporate into multiple projects or workspaces. These external script files are considered top-level artifacts just like the connection and schema artifacts, and they will be managed in the same way. That is, QlikView Expressor will track the code included in these files and where that code is used and you can be guaranteed that the script files will be packaged with your applications during deployment.

Many users of Expressor will also have applications developed using QlikView and will be planning to build applications that combine the two products so as to off-load some of the data transformation processing currently performed in a QlikView load script into an Expressor application. Expressor Datascript, while it includes many versatile functions, does not completely overlap the functionality of the QlikView script expressions. But there is nothing stopping you from implementing equivalent functionality within Expressor, placing your code into a datascript module within a library that can be referenced by many projects.

Note: Expressor Datascript function names, unlike QlikView function names, are case sensitive.

15.1 Exercise: Datascript Implementations of QlikView Expressions

In this exercise, you will create a datascript module that includes several functions that duplicate the processing of QlikView script functions. In developing your implementations, use the following descriptions as your objective, not the possibly more complex descriptions in the QlikView reference manual. The Appendix in this document and the QlikView Expressor product documentation should be used as coding resources.

You might find it easier to develop and test your implementations in a command window running the **datascript** utility and then move them into the Datascript Module.

month (date)

> This function returns the three character abbreviation for the month represented by date. The argument date is a datetime type. How would the code change if the argument date were a string type?

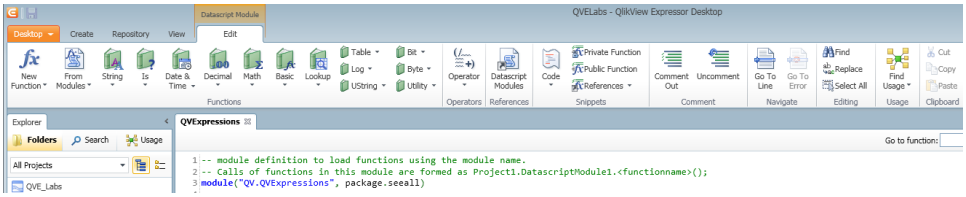
isyeartodate (date, basedate, shift)

> This function returns true if date is an earlier date in the year containing basedate. The year can be offset by shift. Date and basedate are datetime types. Shift is an integer, where the value 0 indicates the year that contains basedate. Negative values in shift indicate preceding years and positive values indicate succeeding years.

mid (s, n1, n2)

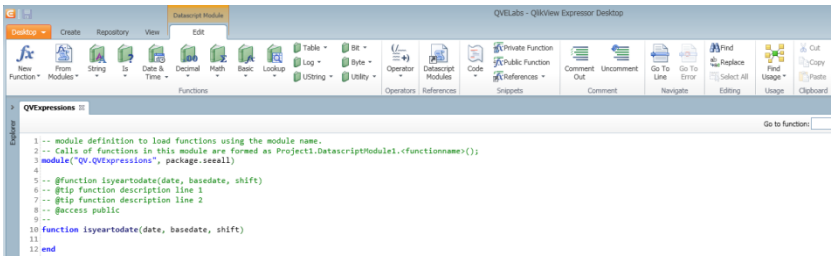
This function returns a substring of string s of length n2 characters starting at character n1. Note, the Expressor Datascript string.substring function returns the substring of string s beginning at character n1 and ending at character n2. The mid function would therefore be an alternative implementation of the substring function.

1. Working in Expressor Desktop, create a new library named **QV**.
2. In the Desktop Explorer panel, expand the library and note the subdirectory name Datascript Moduels.
    a. This is where you will create the files that contain the code you want to write independent of where it will be used.
3. In this library, select the **Datascript Module** subdirectory, right-click, and select **New…** from the popup menu.
    a. Placing your Datascript Module into a library makes it easier to use it with multiple projects and/or workspaces.
4. In the New Datascript Module window, confirm that the module will be added to the library, enter a distinctive name (e.g., QVExpressions), then click **Create**.
    a. The Datascript Module opens in a text editor and the Datascript Module – Edit tab displays in the ribbon bar.
    b. The Datascript Module – Edit ribbon bar tab has many similarities to the ribbon bar that appears when developing rules in that you can quickly include one of the pre-existing QlikView Expressor Datascript functions by simply clicking on the appropriate function class button and selecting the desired function from the drop-down menu. You can also easily include a skeleton for a public or private function that you would like to write.
    c. To give yourself more room to code, collapse the Explorer panel by clicking the  icon in the panel's upper right corner.

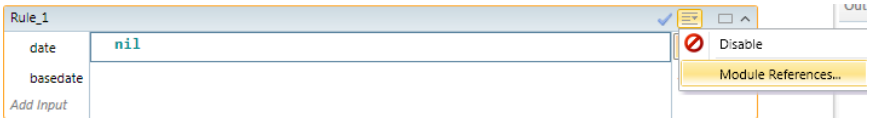


5. The first line in a Datascript Module should be the module declaration, which defines a namespace for your code.
   The namespace is the name of the project or library (without the version number) pre-pended to the name of the Datascript Module.
   The argument package.seeall is required; it makes all of the pre-existing QlikView Expressor Datascript functions accessible to the code you will write.
    a. The function class names that are part of QlikView Expressor Datascript (that is, string, datetime, etc.) are also namespaces.
6. To write a public function, that is, a function that will be visible when you are working with the Rules Editor, click either **Public Function** in the **Snippets** category on the Datascript Module – Edit tab of the ribbon bar or click **New Function – Public Function** in the **Functions** category.
    a. Either approach inserts a comment/annotation into your code.

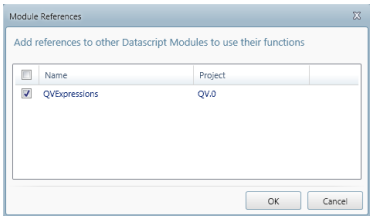
b. Immediately below this comment/annotation, enter your function implementation then flush out the comment details.



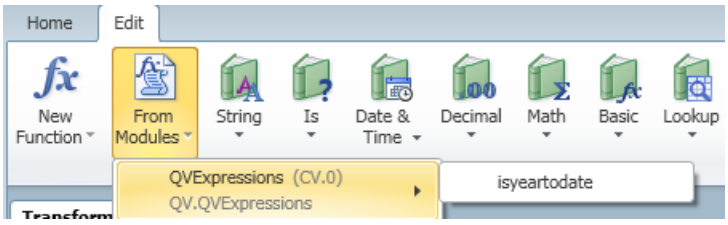
7. Write implementations of the functions described above.
    a. Your implementations may utilize coding you choose to include in a private function.
        i. A private function will be visible only to the code in your Datascript Module not within the Rules Editor.
8. Develop an application to test your logic.
    a. You can develop and test this application in the library named QV.
    b. You will need to create File Connection, Schema, and Dataflow artifacts.
        i. The Dataflow will probably require the Read File, Transform, and Write File operators.
    c. In the Transform operator,
        i. Add an expression rule then add a reference to your module. Click on **Module References…** to open the Module References window.



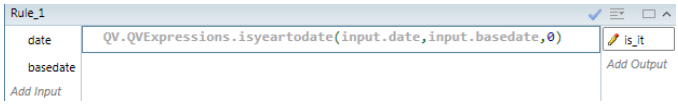
        ii. Select the Datascript Module and click **OK**.



        iii. The code in your module is now available in this rule.
        iv. To use a function from the module, place your cursor into the expression rule then click **From Modules** in the Edit tab of the rules editor ribbon bar.



            1. The function name is prefaced with the module name. Complete the coding by filling in the function arguments.



16 The Read Custom Operator

QlikView Expressor includes many fully implemented input operators such as Read File, Read Table, and SQL Query. While these three operators are suitable for many applications, you may encounter a situation in which you need to access a data resource not supported by these operators. In this situation, the Read Custom operator allows you to create your own input operator. To use this operator, you use QlikView Expressor Datascript to read and parse the data into the individual records emitted by the operator. The operator has one required function and two optional helper functions, but it is likely that you will need to provide implementations for all three functions.

Function Type	Function Name	Description

Mandatory Functions	read	The data processing engine invokes this function repeatedly to produce each record emitted by the operator. Include in the method body QlikView Expressor Datascript that initializes the outgoing record.
		
		This function has no input arguments. It returns a status value, which determines whether invocations of this function continue, and the output record or a message. The status values are defined in the datascript module named expressor.ScriptSupport.
		
		• The QlikView Expressor runtime will continue to invoke this function as long as the previous invocation returns the output record or two return values – the status value expressor.ScriptSupport.OK **and** the output record.
		• Repeated invocations of this function will cease if the previous invocation returns true or the status value expressor.ScriptSupport.FlowComplete.
		• This function may also return the status values expressor.ScriptSupport.SkipRecord or expressor.ScriptSupport.RejectRecord.
		  ○ If SkipRecord is returned, the read function continues to be invoked after skipping the current record.
		  ○ If RejectRecord is returned, the record and optional messages are emitted from the operator's reject port.

Optional Helper Functions	initialize	The data processing engine invokes this function one time before the operator begins to invoke the read function. This function has no arguments or return values. Use this function to set up a connection to your data source, e.g., establish a connection to an FTP server or obtain a handle to a file that will be read during each invocation of the read function.
	finalize	The data processing engine invokes this function one time after the operator has completed emitting all records, that is, after the read function has returned expressor.ScriptSupport.FlowComplete, expressor.ScriptSupport.SkipRemaining, or expressor.ScriptSupport.RejectRemaining. This function has no arguments or return values. Use this function to free any resources obtained during execution of the initialize function.

16.1 Fixed Width File Data

One question that is repeatedly raised on the community forum is how to use Expressor to read fixed width files in which each field always spans a specified and unchanging number of characters and there are no delimiter characters to demarcate the individual fields.

You will note that the schema wizard works only with delimited file data in which one or more characters are used to demarcate the individual fields where any field may contain values of varying width. Consequently, you cannot create a schema that describes a file in which each field has a fixed, unchanging width. To handle data of this type, you need to develop code in the Read Custom operator that obtains and parses each line into its constituent fields.

1. Develop an Expressor dataflow that reads a fixed width file and emits the data as a comma delimited file.
    a. The widths of the fields should be different, for example, the first field might be 3 characters while the second field is 10 characters and a third field is 15 characters.
    b. In the output file, each field should contain only the data value (that is, no padding characters).

In developing this exercise, put the IO related code that will let you read the source file into the operator's `initialize` function. Then in the `read` function, read and parse each line from the file. Since there is no schema to describe the structure of the emitted record, you will need to manually add attributes to the operator's output panel.

- Does the input file include a header row?
    - If so, you don't want to process that row as if it contains a data record.
- How will your code obtain the file system location of the file to process?
    - Do you want to hard code this into your implementation?
        - Is there a way to pass this information into the code from the environment?
            - You know two ways to do this.

16.2 Reading Multiple Files

Frequently, a directory may contain a collection of uniquely named files all of which have the same structure but contain different data. For example, a collection of files with sales data from different store branches. The layout (header names, number and order of fields) of each file is identical but each file contains data from only one store branch.

Obviously the processing to be applied to the data in each file is the same but to process such a collection of files with a dataflow running within Desktop could be quite troublesome as you apparently need to manually change the file name prior to each execution of the dataflow.

There are several ways to approach this requirement within Expressor even though the Read File operator cannot be configured to sequentially process multiple files located in the same directory. All, however, depend on you writing Datascript and using DOS commands to handle the management of the

multiple files. Additionally, the organization of the files may impact your solution, for example, do the files contain a header row and if so is the header row easily distinguishable from the data rows.

1. Develop an Expressor dataflow that reads multiple comma delimited files from a common file system location.
    a. The files must have identical structure; that is, they must all have the same number of fields in the same order.
    b. Your implementation must first obtain a listing of the files in the directory.
        i. The following statement initializes a file handle from which your code can read the names of the files.
        ```
        file_handle = io.popen(string.concatenate("dir /b " ,directory))
        ```
    c. The Windows `type` command will concatenate the contents of a listing of files, similar to the functionality of the UNIX/Linux `cat` command.
        i. The following statement initializes a file handle from which your code can read each line from the concatenated files.
        ```
        file_handle = io.popen(string.concatenate("type " ,file_list))
        ```

In developing this exercise, put the IO related code that will let you read the source file into the operator's `initialize` function. Then in the `read` function, read and parse each line from the file. Since there is no schema to describe the structure of the emitted record, you will need to manually add attributes to the operator's output panel.

- Does the input file include a header row?
    - If so, you don't want to process that row as if it contains a data record.
- How will your code obtain the file system location of the file to process?
- How will your code create the list of files to read in a way that can be passed to the `type` command?

17 Solutions

17.1 is Library Functions

nil	“”	“ “	25	true	“expressor software”
nil	string	string	number	Boolean	string

	is.blank	is.decimal	is.datetime	is.empty	is.future	is.integer	is.null	is.number	is.past
nil	true	false	false	true	false	false	true	false	false
“”	true	false	false	true	false	false	false	false	false
“ “	true	false	false	false	false	false	false	false	false
25	false	false		false		false	false	true	
true		false	false	false	false	false	false	false	false
“expressor software”	false	false	false	false	false	false	false	false	false

What may be confusing is the false value for is.integer(25); this certainly looks like an integer. But it’s not. In QlikView Expressor the decimal, integer and datetime types are defined types that are not part of the underlying Lua implementation. Your code must explicitly create an integer or decimal type from a number. Lua only has a number type, which is used for all types of numeric values.

In using the `is.pattern` function, the pattern to find should be defined using the Lua/Datascript pattern matching syntax. But for this example, since pattern matching has not yet been discussed, simply enter the desired target as the pattern.

```
print(is.pattern(“Expressor Datascript”,”Expressor”))
```

17.2 os Library Functions

All three environment variables are retrieved in the same way with the `os.getenv` function, simply change the name of the variable to retrieve.

```
print(os.getenv(“PATH”))
```

To change the name of a file, use the `os.rename` function. From within the directory that holds the file, you only need to specify the existing and new file names.

```
os.rename(“myfile.txt”,”yourfile.txt”)
```

From another directory, you need to include the path to both the original and renamed files. Note that \ is a special character (the escape character) that must be escaped.

```
os.rename(“C:\\mywork\\myfile.txt”,”C:\\mywork\\yourfile.txt”)
```

17.3 io Library Functions

This illustrates how to create and use a file handle.

```
file_handle = io.open("C:\\mywork\\test.txt","r")
line = file_handle:read("*l")
while line do
  print(line)
  line = file_handle:read("*l")
end
file_handle:close()
```

To write, rather than display the file's contents, you must create a second file handle for the output file.

```
file_handle = io.open("C:\\mywork\\test.txt","r")
file_handle2 = io.open("C:\\mywork\\test2.txt","w")
line = file_handle:read("*l")
while line do
  file_handle2:write(line, "\r\n")
  line = file_handle:read("*l")
end
file_handle2:flush()
file_handle2:close()
file_handle:close()
```

17.4 table Library Functions

Initialize a numerically indexed table with airport abbreviations and print one per line.

```
airports = {"BOS","LHR","AMS","EWR"}
for key,value in ipairs(airports) do
  print(key,value)
end
```

Print a single line with all abbreviations delimited by a comma, pipe or tab.

```
airports = {"BOS","LHR","AMS","EWR"}
print(table.concat(airports,"\t"))
```

Add another airport to the table.

```
airports[#airports+1] = "LAX"

table.insert(airports,"LAX")
```

Sort the table in alphabetical order.

```
table.sort(airports)
```

or

```
table.sort(airports,function() return b>a end)
```

Sort the table in reverse alphabetical order.

```
table.sort(airports,function(a,b) return a>b end)
```

To convert abbreviations into city names requires a string indexed table.

```
airports = {BOS= "Boston",AMS= "Amsterdam",LHR= "London",EWR=
"Newark"}

-- assume that abbreviations are held in a variable named airport
airport = "Bos"

print(airports[string.upper(airport)])
```

If a meaningless abbreviation is supplied, nil is returned. This is an easy check to see if a corresponding entry exists in the table.

```
airports =
  {BOS="Boston",AMS="Amsterdam",LHR="London",EWR="Newark"}

-- assume that abbreviations are held in a variable named airport
airport = "xyz"

print(airports[string.upper(airport)])
```

To use either the abbreviation or the city name as the lookup key into the table, you need to try to compare the supplied entry with both the key and value from the table. Be sure to account for the possibility that the supplied entry may not be properly capitalized.

```
airports =
  {BOS="Boston",AMS="Amsterdam",LHR="London",EWR="Newark"}

airport = "boston"

for key,value in pairs(airports) do
  if string.upper(airport) == key
    or string.title(airport) == value
  then
```

```
    print(value)
    break
  end
end
```

17.5 string Library Functions

The point to consider is that you want to return all content after the first dot is the portion of the email address that follows the `@` sign. Review the difference between the patterns `.+` and `.-`.

```
email= "harry.potter@qlikview.com"
domain = string.match(email, ".+@.-%.(.+)")
print(domain)

email= "harry.potter@qlikview.com.au"
domain = string.match(email, ".+@.-%.(.+)")
print(domain)
```

To remove the single quotation marks you could use the following functions.

- `string.filter`, which will remove the single quotation marks
  - The best approach
- `string.replace`, which will replace the single quotation marks with an empty character
  - An acceptable approach
- `string.allow`, in which you must specify all allowed characters
  - Not a very good approach

The `string.frequency` function will return the desired information.

```
print(string.frequency("Mississippi","ss"))
```

To read a file as one continuous string, you supply the `"*a"` parameter to the `io` read function. Then use the `string.iterate` function to parse this continuous string into separate lines (that is, all content before the control characters indicating the end of a line).

```
file_handle=io.open("myfile.txt","r")
content = file_handle:read("*a")
file_handle:close()

for line in string.iterate(content,"(%C+)") do
  print(line)
end
```

17.6 datetime Library Functions

Use the `datetime.elapse` function to determine your age in days. Note that you will need to use the `string.datetime` function to create a `datetime` value for your birthdate. You can use the `datetime.timestamp` function to retrieve today's date, but then reformat this datetime value to remove the time component.

```
today = datetime.timestamp()
today = datetime.string(today,"M*/D*/CCYY")
today = string.datetime(today,"M*/D*/CCYY")

birthday = string.datetime("1/15/2000","M*/D*/CCYY")

days_old = datetime.elapse(birthday,today,"d")
print(days_old)
```

To determine the minutes and seconds in an 8 hour day, create two `datetime` values that differ by 8 hours. It doesn't matter what date you use; in fact, no date is fine.

```
beginning_time = string.datetime("08:00","HH:MI")
ending_time = string.datetime("17:00","HH:MI")

seconds = datetime.elapse(beginning_time,ending_time,"s")
print(seconds)

minutes = datetime.elapse(beginning_time,ending_time,"i")
print(minutes)
```

The `datetime.string` function will return the day of the week as a string value.

```
birthday = string.datetime("1/15/2000","M*/D*/CCYY")
day_of_week = datetime.string(birthday,"DDDD")
print(day_of_week)
```

Alternatively, the `datetime.moment` function will return a numeric representation of the day of the week that you can then convert into a day name. Think about how the table `days` is initialized.

```
days =
  {"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"}
days[0] = "Sunday"

birthday = string.datetime("1/15/2000","M*/D*/CCYY")
day = datetime.moment(birthday,"w")
day_of_week = days[day]
print(day_of_week)
```

To determine the day of the week 30 days from today, use `datetime.adjust` to add 30 days to today's date. Then use the above approaches to getting the actual day.

```
today = datetime.timestamp()
today = datetime.string(today,"M*/D*/CCYY")
today = string.datetime(today,"M*/D*/CCYY")

days_from_today = datetime.adjust(today,30,"d")
print(datetime.string(days_from_today,"DDDD"))
```

or

```
days =
 {"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"}
days[0] = "Sunday"

today = datetime.timestamp()
today = datetime.string(today,"M*/D*/CCYY")
today = string.datetime(today,"M*/D*/CCYY")

days_from_today = datetime.adjust(today,30,"d")
print(days[datetime.moment(days_from_today,"w")])
```

17.7 utility Library Functions

When using the \ in a quoted string, you must escape it as it is a special character. The escape character is \. So, the path to a directory becomes something like `C:\\myfiles\\data`.

```
utility.store_string("directory", "C:\\myfiles\\data")
directory = utility.retrieve_string("directory")
print(directory)
```

17.8 basic Grouping Functions

Let's assume that the set of boys' names includes John, Bill, and Fred; the set of girls' names includes Sally, Linda, and Abigail.

The approach would be to put the names into a string indexed table where the name is the index and the sex is the value. It's probably better to include both sets of names in a single table.

Using the `decode` function:

```
sexes =
  {John="masculine",Bill="masculine",Fred="masculine",
   Sally="feminine",Linda="feminine",Abigail="feminine"}

name_to_lookup = "John"

sex = decode(
  sexes[name_to_lookup],"masculine","masculine",
    "feminine","feminine","sex unknown")
print(sex)

name_to_lookup = "Abigail"
sex = decode(
  sexes[name_to_lookup],"masculine","masculine",
    "feminine","feminine","sex unknown")
print(sex)

name_to_lookup = "Tom"
sex = decode(
  sexes[name_to_lookup],"masculine","masculine",
    "feminine","feminine","sex unknown")
print(sex)
```

Using the `decision` function:

```
sexes =
  {John="masculine",Bill="masculine",Fred="masculine",
   Sally="feminine",Linda="feminine",Abigail="feminine"}

name_to_lookup = "John"

sex = decision(sexes[name_to_lookup]=="masculine","masculine",
  sexes[name_to_lookup]== "feminine","feminine","sex unknown")
print(sex)

name_to_lookup = "Abigail"

sex = decision(sexes[name_to_lookup]=="masculine","masculine",
  sexes[name_to_lookup]== "feminine","feminine","sex unknown")
print(sex)
```

```
name_to_lookup = "Tom"

sex = decision(sexes[name_to_lookup]=="masculine","masculine",
  sexes[name_to_lookup]== "feminine","feminine","sex unknown")
print(sex)
```

Since there is a possibility that the `string.datetime` function call might raise an error, you need to wrap this call in the `pcall` function. In the following code, the third value will not convert as a suitable format is not included in the `formats` table. This additional format can be easily handled by adding a new format to the `formats` table.

```
formats = {"M*/D*/CCYY","CCYY-M*-D*","MMM D*, CCYY"}

date_to_test = "2000-1-15"
success = false
for key,format in ipairs(formats) do
  success,date = pcall(
    string.datetime,date_to_test,format)
  if success then
    print(datetime.string(date))
    break
  end
end
if not success then print("couldn't convert date") end

date_to_test = "2000-1-15"
success = false
for key,format in ipairs(formats) do
  success,date = pcall(
    string.datetime,date_to_test,format)
  if success then
    print(datetime.string(date))
    break
  end
end
if not success then print("couldn't convert date") end

date_to_test = "January 15, 2000"
success = false
for key,format in ipairs(formats) do
  success,date = pcall(
    string.datetime,date_to_test,format)
  if success then
```

```
    print(datetime.string(date))
    break
  end
end
if not success then print("couldn't convert date") end
```

17.9 Datascript Implementations of QlikView Expressions

```
-- module definition to load functions using the module name.
-- Calls of functions in this module are formed as QV.QVExpressions.<functionname>();
module("QV.QVExpressions", package.seeall)

-- @function month(date)
-- @tip Returns the 3 character month name
-- @tip Argument is datetime type
-- @access public
--
function month(date)
  -- the datetime.string function has an option to return only the 3 character name
  return datetime.string(datetime,"MMM")
end

-- @function isyeartodate(date, basedate, shift)
-- @tip Determines whether date is in the same year as basedate
-- @tip shift alters basedate's year
-- @tip date and basedate are datetime types; shift is an integer
-- @access public
--
function isyeartodate(date, basedate, shift)
  -- adjust basedate as per shift
  basedate = datetime.adjust(basedate,shift,"y")
  -- determine basedate's year
  year = tostring(datetime.moment(basedate,"y"))
  -- January 1st datetime
  starting_date = string.datetime(string.concatenate("01/01/", year),"M*/D*/CCYY")

  return (starting_date <= date) and (date <= basedate)
end
```

```
-- @function mid(s, n1, n2)
-- @tip Returns substring of s or length n2
-- @tip Starting in position n1
-- @access public
--
function mid (s, n1, n2)
  return string.substring(s,n1,n1+n2-1)
end
```

17.10 Fixed Width File Data

Before you can begin, you need to create a file that includes fixed width data. For example, put the following data into a file.

Note that the first field is left padded to width 2 while the remaining fields are right padded (15, 20, 25 characters) and each line ends with a return/line feed.

```
 1George        Washington          none
 2John          Adams               Federalist
 3Thomas        Jefferson           Democratic-Republican
44Barack        Obama               Democratic
```

Since you know the lengths of each field, the most straight forward solution is to read each line as a single field and then use `string.substring` to parse into individual fields. Be sure to also use `string.trim` to remove the padding characters. But this is not a very useful approach as you will need to completely rewrite the coding for a different input file. A better approach is to develop a reusable function within a datascript module and provide specifics about the file structure to this function as an argument.

The following implementation takes such an approach. To process a different file, only the code in the `initialize` function will require modification.

Create a datascript module with a function implementation similar to the following. Note that the name of the project containing this module is ReadCustom and that the module's name is Parse, but those assignments are at the discretion of the developer.

Note how the code in the parse function marches across the input line using the field width information in the argument `fields`. After each field is extracted from the line, those characters are removed from the line so that the all calls to the `string.substring` function can start from position one.

Parse

```
-- module definition to load functions using the module name.
-- Calls of functions in this module are formed as ReadCustom.Parse.<functionname>();
module("ReadCustom.Parse", package.seeall)

-- @function parse(line,fields)
-- @tip pass the line to parse as the first argument
-- @tip pass a table containing attribute names & widths as the second argument
-- @access public
--
function parse(line,fields)
  output = {}

  for k,v in ipairs(fields) do
    output[v[1]] = string.trim(string.substring(line,1,v[2]))
    line = string.substring(line,v[2]+1)
  end

  return output
end
```

Then in the Read Custom operator, use the `initialize` function to set up the handle to the input file and to initialize the `fields` table. The code in the `read` function is then quite straight-forward and will not need editing when processing an input file with a different composition. Note how the location and name of the file to process is obtained using the `utility.retrieve_string` function. You must set this information using the `utility.store_string` function prior to running the dataflow.

Read Custom 1

```
-- @datascript module dependency
require("ReadCustom.0.Parse")
require "expressor.ScriptSupport"

file_handle = nil
fields = {}

function initialize()
  file_handle = io.open(utility.retrieve_string("file"),"r")

  fields[1] = {"Field1",2}
  fields[2] = {"Field2",15}
  fields[3] = {"Field3",20}
  fields[4] = {"Field4",25}
end;

function read()
  line = file_handle:read("*l")
  if line then
    return expressor.ScriptSupport.OK,ReadCustom.Parse.parse(line,fields)
  else
    return expressor.ScriptSupport.FlowComplete
  end
end;

function finalize()
  file_handle:close()
end
```

17.11 Reading Multiple Files

Place several files with identical structure but different content into the same file system location. Then, in the Read Custom operator, enter code similar to the following. Note how the Windows `dir` command is used to obtain a listing of the file names in the source directory and how the `table.concat` function turns a numerically indexed table into a string that can be passed to the Windows `type` command.

If the source files include header rows, which you probably don't want to emit from the operator, your code will need to use the `string.match` function to determine if a line contains text that matches the header row content. If so, you skip that row by returning the expressor.ScriptSupport.SkipRecord value.

Appendix – QlikView Expressor Datascript Functions

The functions you will most likely use are in the `basic`, `datetime`, `io`, `is`, `os`, `string`, `table`, and `utility` libraries. In addition, QlikView Expressor Datascript includes the standard arithmetic, relational, and logical operators that are typical of other programming languages.

Basic Functions

Many of these functions are used to convert a value of one type into another type and will not be reviewed in this document. The functions ipairs, next, and pairs operate on QlikView Expressor Datascript tables and will be detailed in a later section of this document. Within the scope of this training course, the following basic functions are important.

Function	Purpose
decision	This function can be used to create a single statement that implements the same sort of processing as a case statement.
decode	This function can be used to create a single statement that implements the same sort of processing as an if..then..else or an if..elseif..else statement.
ipairs	This function returns an iterator over a numerically indexed QlikView Expressor Datascript table.
pairs	This function returns an iterator over any QlikView Expressor Datascript table.
pcall	This function invokes a function in protected mode, returning true and the function's return values if execution completes normally or false and an error message if execution did not complete normally.
select	This function can be used to select specific return values from a function.
todecimal	Converts a number or numeric string to a decimal type.
tointeger	Converts a number or numeric string to an integer type.

decision function

usage	`decision(comp1, ret1 [,comp_n, ret_n]* [,default])`	
arguments	Comp1	Required logical expression.
	ret1	The return value associated with the required logical expression.
	comp_n, ret_n	Optional additional logical expressions and associated return values.
	default	Optional default return value.
return	true and return value(s) from function or false and error message	

decode function

usage	**decode(val1, val2, ret2 [, val_n, ret_n]* [,default])**	
arguments	val1	The value to be compared.
	val2	Required comparison value.
	ret2	Return value associated with val2.
	val_n, ret_n	Optional comparison values and associated returns.
	default	Optional default return value.
return	true and return value(s) from function or false and error message	

pcall function

usage	**pcall(function [,arguments])**	
arguments	function	A string containing the name of the function to execute.
	arguments	A comma separated list of the arguments to the function.
return	true and return value(s) from function or false and error message	

select function

usage	**select(index, …)**	
arguments	index	The first argument to return.
	…	Function or comma separated listing of arguments to evaluate.
Return	return value at index and	

For example: `print(select(3,"a","b","c","d"))` prints c and d, while enclosing the select function in parentheses limits the return to the value at index.

`print((select(3,"a","b","c","d")))` prints just c

Datetime Functions

Within the scope of this training course, the following datetime functions are important.

Function	Purpose
adjust	Operates on a datetime value returning that value adjusted by a specified interval.
elapse	Operates on two datetime values returning the difference between them.
moment	Operates on a datetime value returning a specific element (seconds, minutes, hours, day, month, year, century).
string	Operates on a datetime value returns a string representation.

adjust function

usage	`datetime.adjust(value, interval [,format [,exact]])`	
arguments	value	The datetime value to adjust.
	interval	The adjustment to be applied.
	format	The interpretation of the adjustment interval. format (case insensitive) / interpretation: none – seconds s – seconds i – minutes h – hours d – days y – years c – centuries
	exact	If false (the default), intervals are calculated using a 365.25 day year. If true, intervals are calculated using a 365 day year.
return	datetime	

elapse function

usage	`datetime.elapse(value1, value2 [,format])`	
arguments	value1	The starting datetime value.
	value2	The ending datetime value.
	format	The units in which to express the differential.

format (case insensitive)	interpretation
none	seconds
s	seconds
i	minutes
h	hours
d	days
y	years
c	centuries

Note that months (m) is not a valid format.

return	number

moment function

usage	`datetime.moment(value [,format])`
arguments	value: The datetime value to analyze.
	format: The units in which to express the differential.

format (case insensitive)	interpretation
none	seconds
s	seconds
i	minutes
h	hours
d	days
j	Julian day (leap year has 366 days)
w	day of week (Sunday is 0)
m	month
y	year
c	century

return	number

string function

usage	`datetime.string(value [,format])`
arguments	value: The datetime value to convert into a string.
	format: The format of the string representation. Default is `CCYY-MM-DD HH24:MI:SS`.

format	interpretation

HH24	hours in 24 hour format
H*24	hours in 24 hour format; single digit hour formatting when appropriate
HH12	hours in 12 hour format
H*12	hours in 12 hour format; single digit hour formatting when appropriate
HH	hours in 24 hour format
H*	hours in 24 hour format; single digit hour formatting when appropriate
MI	minutes
SS	seconds
S[ssssss]	fractional seconds
AM or PM	used with HH or HH12 to indicate whether hour values are AM or PM; only valid if a full time format, including fractional seconds, is specified; this value is included in the output
DD	day in numeric format
D*	day specified as either one or two digits format pattern must be delimited, i.e., MM-D*-CCYY or MM/D*/CCYY, not MMD*CCYY; valid format delimiters are space, hyphen, forward slash, comma and period
D?	invalid day specification accepted; converts the day to either 01 or the last day of the month based on the input value
DM	allows processing of mixed day/month, giving precedence to day; used in conjunction with the MD format
DDD	day of week abbreviated (e.g., MON)
DAY	day of week abbreviated (e.g., MON)
DDDD	day of week in long format (e.g., MONDAY)
DDAY	day of week in long format (e.g., MONDAY)
JJJ	Julian day of year
MM	month in numeric format
M*	month specified as either one or two digits; format pattern must be delimited, i.e., M*-DD-CCYY or M*/DD/CCYY, not

			M*DDCCYY; valid format delimiters are space, hyphen, forward slash, comma and period
		M?	invalid month specification accepted
		MD	allows processing of mixed month/day, giving precedence to month; used in conjunction with DM format
		MMM	month in short format (e.g., JAN)
		MMMM	month in long format (e.g., January)
		YY	years
		YNN	Forces a century designation anchored to NN; In a date field, a two character year is interpreted as the current century if less than NN and the previous century if greater than NN
		CC	century
return	string		

IO Functions

Within the scope of this training course, the following IO functions are important.

Function	Purpose
io.open	Opens a file for reading, writing, etc.
file_handle:close	Closes a file.
file_handle:read	Reads a file's contents.
file_handle:lines	Returns an iterator function for reading a file line by line.
file_handle:write	Writes to a file.
file_handle:flush	Flushes data to a file.
io.popen	Starts a program in a separate process.

open

usage	`io.open(fn [,mode])`	
arguments	fn	A string containing the name of the file.
	mode	A string containing the mode. "r", read; "w", write; "a", append; "r+", update-preserve, "w+", update-erase; "a+", update-append
return	file handle used to invoke other IO functions	

close

usage	`file_handle:close()`	
arguments	none	
return	none	

read

usage	`file_handle:read(format)`	
arguments	format	A string specifying what to read. "*n", number; "*a", entire file (returns "" at end of file); "*l", a line (returns nil at end of file); n, string of up to n characters (returns nil at end of file)
return	varies	

lines

usage	`file_handle:lines()`	
arguments	none	
return	iterator function for reading file line-by-line	

write

usage	`file_hande:write(values)`	
arguments	values	Values to write to file. Values may be numbers or strings. No separators are added between values
return	none	

flush

usage	`file_handle:flush()`	
arguments	none	
return	none	

popen

usage	`io.popen(prog [,mode])`	
arguments	prog	A string containing the name of a program/command to run.
	mode	A string containing the mode. “r”, read; “w”, write
return	file handle used to invoke other IO functions	

IS Functions

Within the scope of this training course, the following is functions are important.

Function	Purpose
decimal, datetime, integer, number, string	Return true if the value being tested is of the corresponding type.
blank	Returns true if the value being tested consists entirely of space characters.
empty	Returns true for a nil value, zero numeric value, or empty string value.
future, past	Returns true if datetime value is in the future or past.
null	Returns true if value is nil.
pattern	Returns a character pattern if it is present in the value being tested; otherwise returns nil.

blank, decimal, datetime, empty, future, integer, null, number, past, string functions

usage	`is...(value)`	
arguments	value	The value to test
return	boolean	

pattern

usage	`is.pattern(value, pattern[, begin])`	
arguments	value	The value to test.
	pattern	The pattern to find.
	begin	Optional start position at which to begin the search. Default is 1. Negative values are offsets from the right end of value.
return	string	

OS Functions

Within the scope of this training course, the following OS functions are important.

Function	Purpose
execute	Executes a command returning a system dependent status code.
getenv	Returns a string with the value of an environment variable or nil if the variable does not exits.
remove	Deletes a file.
rename	Renames a file.

execute

usage	`os.execute(cmd)`	
arguments	cmd	A string containing the command to execute.
return	number	

getenv

usage	`os.getenv(var)`	
arguments	var	String containing the name of an environment variable.
return	string	

remove

usage	`os.remove(fn)`	
arguments	fn	String containing name of the file to delete.
return	In case of error, nil and error description	

rename

usage	`os.rename(of, nf)`	
arguments	of	String containing current name of file.
	nf	String containing new name of file.
return	In case of error, nil and error description	

String Functions

Within the scope of this training course, the following string functions are important. These functions all operate on a string value, generally returning an altered string.

Function	Purpose
allow	Returns a string containing only the allowed characters.
concatenate	Operates on a list of values returning a concatenated string.
datetime	Converts a string representation of a datetime into an unformatted datetime type.
filter	Returns a string from which the filtered characters have been removed.
find	Returns the starting, ending, and optional capture of a specified character pattern.
frequency	Returns the number of times a specified character pattern is found.
iterate	Returns all occurrences of a specified character pattern.
length	Returns the length of a string.
match	Returns, if present, the characters corresponding to a specified character pattern.
replace	Returns a string with all occurrences of a specified character sequence replaced with another character sequence.
substring	Returns a substring of the string.
trim	Removes space characters from both ends of the string.

allow function

usage	`string.allow(value, allowed)`	
arguments	value	String value to modify
	allowed	Subset of characters to return
return	string	

concatenate function

usage	`string.concatenate(value [,value_n])`	
arguments	value	First value to be concatenated
	value_n	Additional values to be concatenated
return	string	

usage `string.datetime(value [,format])`

arguments

value — String to be converted into a datetime.

format — The format of the string representation. The format is optional only when the string representation has the default format. Default format is `CCYY-MM-DD HH24:MI:SS`.

format	interpretation
HH24	hours in 24 hour format
H*24	hours in 24 hour format; single digit hour formatting when appropriate
HH12	hours in 12 hour format
H*12	hours in 12 hour format; single digit hour formatting when appropriate
HH	hours in 24 hour format
H*	hours in 24 hour format; single digit hour formatting when appropriate
MI	minutes
SS	seconds
S[ssssss]	fractional seconds
AM or PM	used with HH or HH12 to indicate whether hour values are AM or PM; only valid if a full time format, including fractional seconds, is specified; this value is included in the output
DD	day in numeric format
D*	day specified as either one or two digits format pattern must be delimited, i.e., MM-D*-CCYY or MM/D*/CCYY, not MMD*CCYY; valid format delimiters are space, hyphen, forward slash, comma and period
D?	invalid day specification accepted; converts the day to either 01 or the last day of the month based on the input value
DM	allows processing of mixed day/month, giving precedence to day; used in conjunction with the MD format
DDD	day of week abbreviated (e.g., MON)
DAY	day of week abbreviated (e.g., MON)

	DDDD	day of week in long format (e.g., MONDAY)
	DDAY	day of week in long format (e.g., MONDAY)
	JJJ	Julian day of year
	MM	month in numeric format
	M*	month specified as either one or two digits; format pattern must be delimited, i.e., M*-DD-CCYY or M*/DD/CCYY, not M*DDCCYY; valid format delimiters are space, hyphen, forward slash, comma and period
	M?	invalid month specification accepted
	MD	allows processing of mixed month/day, giving precedence to month; used in conjunction with DM format
	MMM	month in short format (e.g., JAN)
	MMMM	month in long format (e.g., January)
	YY	years
	YNN	Forces a century designation anchored to NN; In a date field, a two character year is interpreted as the current century if less than NN and the previous century if greater than NN
	CC	century
return	datetime	

filter function

usage	`string.filter(value, filter)`	
arguments	value	String value to modify
	filter	Subset of characters to remove
return	string	

find function

usage	**string.find(value, pattern [, begin [, offset]])**	
arguments	value	The string value to examine.
	pattern	Character pattern to find
	begin	Optional starting character in value. If negative, offset search from right end of value and search from left to right.
	offset	If true, turn off character class capabilities in pattern and the function does a basic find substring operation.
return	Integer, integer [, string]	

iterate function

usage	**string.iterate(value, pattern)**	
arguments	value	The string to examine
	pattern	A character pattern to find in the value
return	function	

Since the iterate function returns an iterator, you must capture the possible multiple return character strings in a numerically indexed table. To determine if the iterator function returned any captures, check the length of the table.

```
return_strings = {}
for ret in string.iterate(value, pattern) do
  return_strings[#return_strings+1] = ret
end
```

frequency function

usage	**string.frequency(value, pattern)**	
arguments	value	String value to examine
	pattern	A character pattern to find in the value
return	number	

length function

usage	`string.length(value)`	
arguments	value	The string to examine
return	integer	

match function

usage	`string.match(value, pattern [, begin])`	
arguments	value	The string value to examine
	pattern	Character pattern to find
	begin	Optional starting character position. If negative, offset search from right end of value and search from left to right.
return	String	

replace function

usage	`string.replace(value, old, new[,count])`	
arguments	value	String value to modify
	old	Character sequence to be replaced
	new	Character sequence to be inserted
	count	Number of times to replace the character string old
return	string (the revised string), integer (number of replacements)	

substring function

usage	`string.substring(value [, start [, end]])`	
arguments	value	String value to modify
	old	Optional positional offset into value; default is 1. If negative, offset from right end of value.
	new	Optional positional offset of last character to return. Default is the last character of value.
return	string	

trim function

usage	`string.trim(value)`	
arguments	value	String value to modify
return	string	

Table Functions

Within the scope of this training course, the following table functions are important.

Function	Purpose
concat	Concatenates the values of a numerically indexed table into a single string.
insert	Inserts an element into a numerically indexed table.
remove	Removes an element from a numerically indexed table.
sort	Sorts a numerically indexed table.
#	Returns the number of elements in a numerically indexed table.

concat

usage	`table.concat(table [, separator [, first [, last]]])`	
arguments	table	A numerically indexed table where all values are numbers or strings.
	separator	The character to insert between each value extracted from the table. The default separator is a space character.
	first	The index of the first element to extract from the table. Default is 1.
	last	The index of the last element to extract from the table. Default is the last element in continuous numerical order.
return	string	

insert

usage	`table.insert(table, [position,] value)`	
arguments	table	A numerically indexed table into which to insert an element.
	position	The index at which to insert the element. Default is at the end of the table.
	value	The value to insert
return	The table is modified in place.	

remove

usage	`table.remove(table [,position])`	
arguments	table	A numerically indexed table from which to remove an element.
	position	The index at which to remove the element. Default is at the end of the table.
return	file handle used to invoke other IO functions	

sort

usage	`table.sort(table [,order])`	
arguments	table	A numerically indexed table to sort.
	order	A function that takes two arguments and returns true when the table element represented by the first argument should come before the table element represented by the second element.
return	The table is modified in place.	

For example, this order function, where a and b represent values within each table element, will result in the table being sorted descending.

```
table.sort(table, function(a,b) return (a>b) end)
```

While this order function will result in the table being sorted ascending.

```
table.sort(table, function(a,b) return (a<b) end)
```

#

usage	`#table`
arguments	none
return	The number of elements in a numerically indexed table.

Utility Functions

Within the scope of this training course, the following utility functions are important.

Function	Purpose
store_binary	Stores a value of the specified data type in the persistent store.
store_boolean	
store_datetime	
store_decimal	
store_double	
store_integer	
store_number	
store_string	
retrieve_binary	Retrieves a value of the specified data type from the persistent store.
retrieve_boolean	
retrieve_datetime	
retrieve_decimal	
retrieve_double	
retrieve_integer	
retrieve_number	
retrieve_string	

store_...

usage	`utility.store_...(name,value)`	
arguments	name	A string literal or a reference to a string variable providing the name under which to store the value.
	value	The value or a reference to a variable containing the value to store.
return	none	

retrieve_...

usage	`utility.retrieve_...(name)`	
arguments	name	A string literal or a reference to a string variable providing the name of the value to retrieve.
return	the retrieved value	

QlikView Expressor Datascript Pattern Matching Syntax

Several of the string functions utilize a pattern to isolate character strings within a larger string. A character pattern could be as simple as a few characters enclosed within quotation marks, such as, “expressor”. Alternatively, a pattern could be a little more cryptic, for example, all alphanumeric characters before the @ character. The QlikView Expressor Datascript Pattern Matching Syntax lets you develop patterns that can identify any character string. In order to work with patterns, you must understand the following concepts.

Character Class

A character class is used o represent a set of characters. The following character combinations are allowed when describing a character class.

- Any keyboard character represents itself.
    - The characters `^$()%.[]*+-?` are “magic” cracters that cannot directly represent themselves. These characters must be escaped with a preceding `%` character.
    - The `%` escape character may be placed before any non-alphanumeric character (e.g., punctuation marks, slashes, or the pipe character) to ensure that no special interpretation is attached to the character.
- A dot (period) represents all characters.
- %a represents all letters.
    - %A represents the complement of %a.
- %c represents all control characters.
    - %C represents the complement of %c.
- %d represents all digits.
    - %D represents the complement of %d.
- %l represents all lower case letters.
    - %L represents the complement of %l.
- %p represents all punctuation characters.
    - %P represents the complement of %p.
- %s represents all space characters.
    - %S represents the complement of %s.
- %u represents all upper case letters.
    - %U represents the complement of %u.
- %w represents all alphanumeric characters.
    - %W represents the complement of %w.
- %x represents all hexadecimal digits.
    - %X represents the complement of %x.
- %z represents the character that represents zero value.
    - %Z represents the complement of %z.

Set

A character class that includes a union of characters indicated by enclosing the characters in square brackets [ ] is referred to as a set. All character combinations can also be used as components in a set.

- You specify a range of characters in a set by separating the end characters with a dash. For example: [1-10]
- The complement of a set is represented by including the ^ character at the beginning of the set. For example: [^1-10]

Pattern Item

A pattern item can be represented by:

- A single character class, which matches any single character in the class.
- A single character class followed by *, which matches zero or more repetitions of characters in the class. These repetition items always match the longest possible sequence.
- A single character class followed by -, which matches zero or more repetitions of characters in the class. These repetition items always match the shortest possible sequence.
- A single character class followed by +, which matches 1 or more repetitions of characters in the class. These repetition items always match the longest possible sequence.
- A single character class followed by ?, which matches zero or 1 occurrence of a character in the class.

Pattern

A pattern is a sequence of pattern items.

- ^ at the beginning of a pattern anchors the match at the beginning of the string.
- $ at the end of a pattern anchors the match at the end of the string.
- A pattern cannot contain embedded zeros; use %z instead.

Captures

A pattern can contain sub-patterns enclosed in parentheses, which describe “captures.” When a match succeeds, the substrings of the analyzed string that match the captures are stored (captured) for future use.