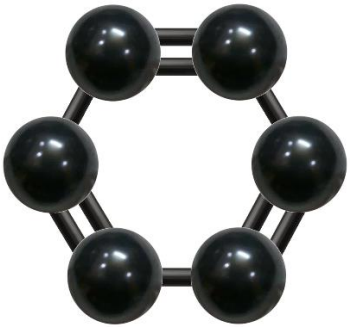


Carbon Scripting

Version 10 September 2022

This document describes how to use the Carbon DLLs (dynamic link libraries) with C# scripting.

OVERVIEW

The Carbon libraries implement Red Centre Software's pluggable crosstab engine, available as Nuget packages, DLLs and code libraries which can be called from many languages and development environments. Some possible combinations are

- Text editor for writing C# Interactive scripts (*.csx), and a Command Line or PowerShell console to call the script compiler as >rcsx.exe <script file>
- Visual Studio using C# or VB.Net
- Excel or any MS Office VBA
- Visual Studio Code using C#, C# Interactive or VB.Net
- Jupyter Notebooks using Python, C# or C# Interactive

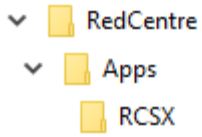
This document adopts the first option: a text editor for writing CSX scripts and the Command Line console for execution. This is often abbreviated as CLI, for *Command Line Interface*. It usually appears as a black background with white text  but for this document's screen shots the background is light grey with black text.

C# Interactive is a simplified implementaion of C#, and is usually adequate for our purposes. For a full C# example project see \RedCentre\Jobs\Scripted\SAV\VSX\CS_Net6.

All the work is done by simple batch files. The batch file calls the Red Centre script compiler/executor rcsx.exe with the *.csx script file as the first argument.

The official and direct way to execute a CSX file is by csi.exe (C Sharp Interactive, part of the Roslyn compiler suite) but unfortunately csi.exe does not yet work with DotNet Core (DotNet5, 6 and later), and since Microsoft will not commit to a timeline for addressing this, we wrote our own wrapper for the Net6 CSScript control (*rcsx.exe,* for Run CSX). If you are familiar with C# or C# Interactive, note that rcsx.exe has the most frequent System references, and all the Carbon references, pre-referenced with their associated usings statements. See Appendix 6 for details.

The combined package is delivered within RedCentre.zip with this structure:



The Carbon DLLs are stored in \Apps, and rcsx.exe and its support DLLs are in \Apps\RCSX.

For now, to avoid file path complications, unzipping directly to your C: drive is preferred.

Case Sensitivity

Unlike VB, CSX script keywords are case sensitive. For example,

```
console.writeline("Hello Carbon")
```

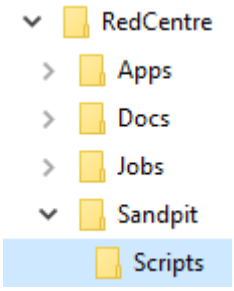
will not be recognized. Instead, capitalise as

```
Console.WriteLine("Hello Carbon")
```

Script names and CLI or PowerShell commands are not case sensitive.

Script Examples

The scripts covered by this document are in \RedCentre\Sandpit\Scripts.



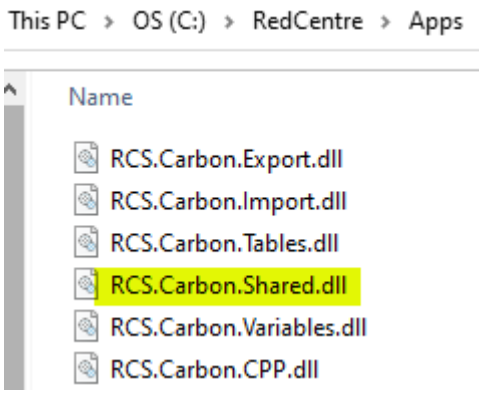
For other CSX examples, see \RedCentre\Jobs\Scripted\SAV\Scripts and \RedCentre\Jobs\Demo\Scripts. The SAV job has a Process.csx script which is intended as a paradigm for full data processing – import, constructions, tables, export. The Demo and SAV job scripts should be transparent after working through the Sandpit examples here.

If you are already familiar with Ruby scripting, and want to just dive in, browsing \RedCentre\Jobs\Scripted\SAV\Scripts\Process.csx and its loaded files Tables.csx, Variables.csx and ImportExport.cs will tell you most of what you need to know.

CARBON

Carbon is our collective name for the DLLs in \RedCentre\Apps. As carbon is an essential element for all life on earth, so by analogy the Carbon DLLs are elemental for cross tabulation.

The Carbon DLLs are

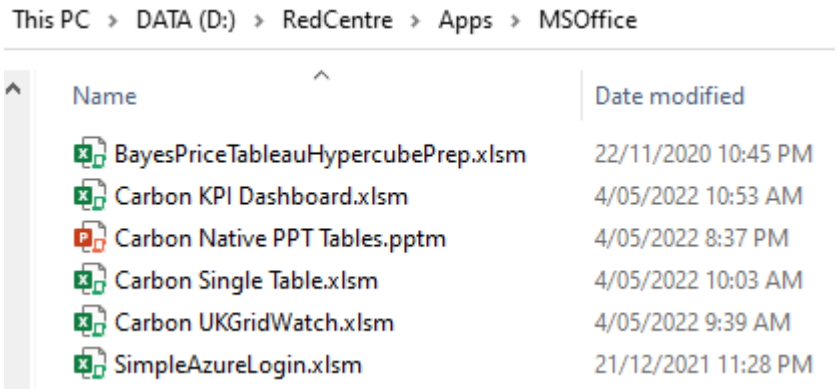


RCS.Carbon.Shared.dll handles logins and other common functions.

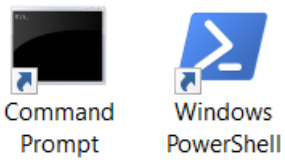
Managed vs Unmanaged Native

The first five DLLs (Export, Import, Tables, Shared, Variables) are written in C#. The last, RCS.Carbon.CPP.dll, is, as the name suggests, written in C++. The C# DLLs are *managed*. Managed C# DLLs are generally more tolerant of programmatic errors or misuse than unmanaged native C++. See https://www.developer.com/microsoft/c-sharp/managed-unmanaged-native-what-kind-of-code-is-this/ for details. The unmanaged C++ DLL is needed for MS Office (whose architecture predates C#) and for any Windows or server applications which need maximum speed and minimal footprint. All CSX scripts call the managed C# DLLs.

For examples of how to use RCS.Carbon.CPP.dll, see the Excel and PowerPoint example applications in \RedCentre\\Apps\MSOffice:



Getting Started



You can use either the Command Prompt or Windows PowerShell to run scripts. The Command Prompt is the old CLI (cmd.exe) and PowerShell is the modern replacement. They each have

their idiosyncrasies. See Appendix 1 for PowerShell and Appendix 2 for Command Prompt to cover any setup issues.

All the script examples have the *.csx* extension, which is the official extension for C# Interactive scripts. You need one batch file to connect your scripts and the script compiler. That is *rcsx.ps1* for PowerShell and *rcsx.bat* for Command Prompt. They each have just one line:

```
rcsx.ps1: \RedCentre\Apps\RCSX\rcsx.exe $args[0] $args[1] $args[2] $args[3] $args[4]

rcsx.bat: @\RedCentre\Apps\RCSX\rcsx.exe %1 %2 %3 %4 %5
```

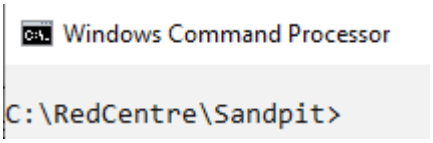
The root drive of your current working directory is assumed. The first argument is either -? for help or the script file name. The other arguments are optional flags. The leading @ suppresses echoing.

If your \Apps\RCSX folder is in a different location, you will need to modify the path to rcsx.exe accordingly. In either case all your commands will be of the form

PowerShell, rcsx.psi:	>.\rcsx myscript.csx [optional flags]
CommandPrompt, rcsx.bat:	> rcsx myscript.csx [optional flags]

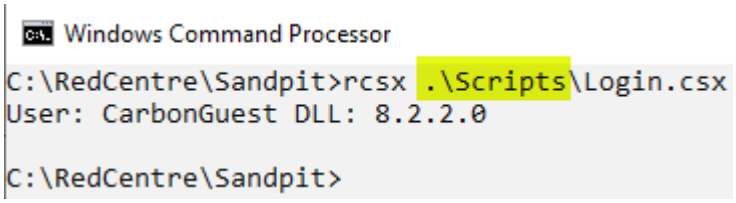
See Appendix1 for why PowerShell needs the leading .\ and how to avoid that if you prefer.

The examples in this document use <root>:\RedCentre\Sandpit\ as the working directory.

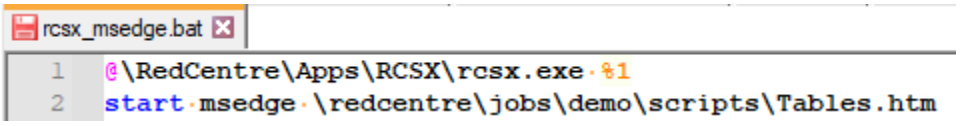


To access the scripts in the *Scripts* directory we need *.\Scripts* to indicate the script is in a sub-directory of the current directory.

That is:



You can of course customise the BAT or PS1 files. This variation calls MsEdge to display the tables generated by the %1 argument:

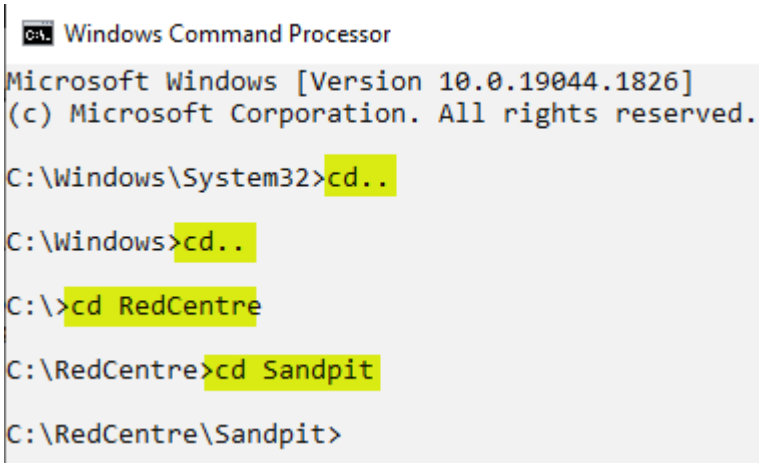


Run a Script

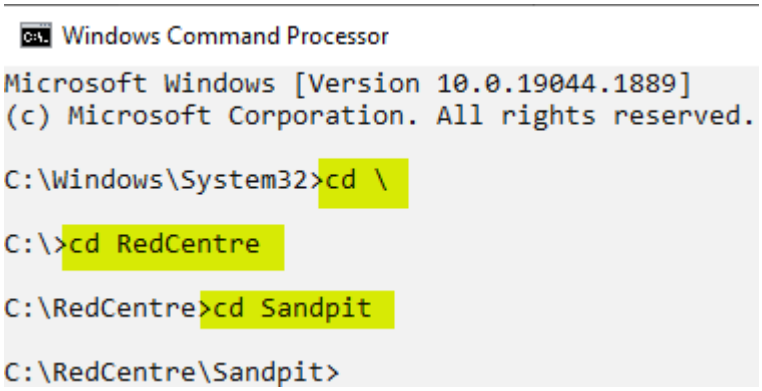
Here is a step by step to run your first script.

1. Start your Command Line console (see Appendix 2 for instructions)

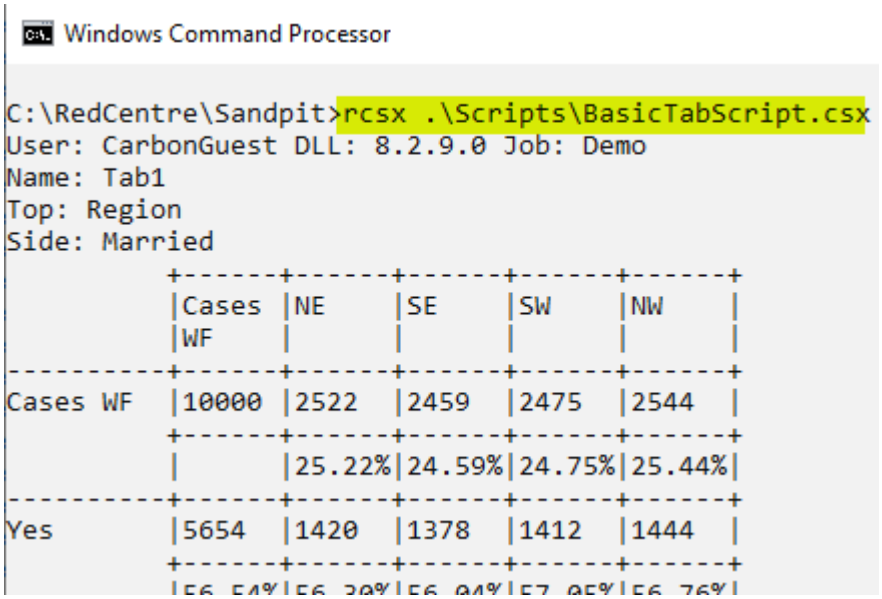
2. Navigate to \RedCentre\Sandpit using the cd command



You could use *cd* \ to jump to the root from any sub-folder, avoiding the second *cd..* above:



3. Type *rcsx .\Scripts\BasicTabScript.csx* and press Enter



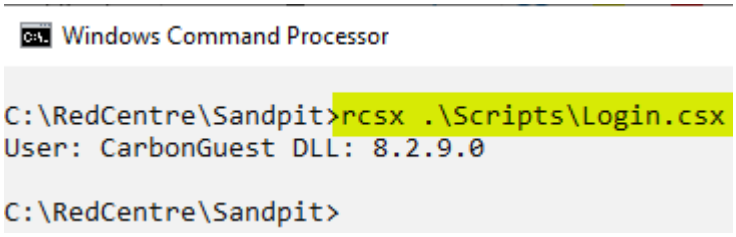
For writing/testing operations scripts, you keep the *.csx file open in a text editor, save at each change, then use the arrow-up key in the Console window to restore the last command, then finally press Enter to rerun the script.

Licensing

To use the Carbon library, you need a licence. Your normal Ruby licence will work, or you can use the standard Carbon Guest licence as in the examples here. Every script must execute at least the first two lines below.

Login.csx

```
var eng = new CrossTabEngine();
var login = await eng.LoginId("16499372","C6H12O6");    // guest licence
Console.WriteLine($"User: {login.Name} DLL: {eng.Version}");
```



CrossTabEngine is an object defined in RCS.Carbon.Tables.dll. You create one of these (here referenced as *eng*) and call its *LoginId* method, passing your user Id and password. The *await* modifier is because the function is asynchronous and waits for the return value before proceeding. The licence call could fail for many reasons (internet down, wrong credentials, licence expired etc). If this happens the script aborts with an error message. The credentials here are for the Carbon guest licence which anyone can use.

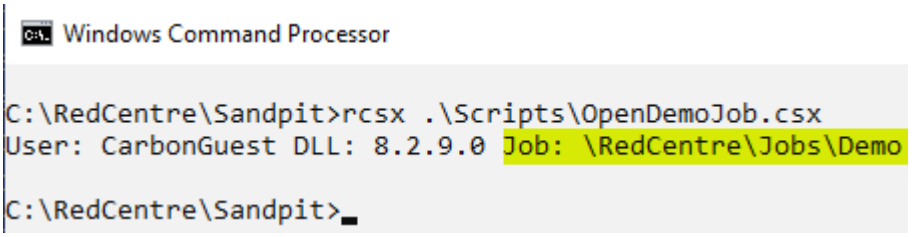
Console.WriteLine shows how to give progress messages. *Console.WriteLine* can take a compound string. The leading *$* means the text will be interspersed with objects – could be any of your variables or values. Anything inside {} is treated as an object reference and turned into a string. In this case the LicenceSummary object, referred to as *login*, has a member *Name*, referred to as *login.Name*, and the engine has a member called *Version*, referred to as *eng.Version*.

Select a Job

The next step is to select a job. Local jobs are identified solely by their folder path.

OpenDemoJob.csx

```
var eng = new CrossTabEngine();
var login = await eng.LoginId("16499372","C6H12O6");    // guest licence

var jobdir = @"\RedCentre\Jobs\Demo";
eng.OpenJob(jobdir);
Console.WriteLine($"User: {login.Name} DLL: {eng.Version} Job: {jobdir}");
```



All the C family languages (C, C#, C++, CSI, Java, JavaScript) use backslash as the escape character, and \\ for a backslash itself. All those \\ in a long path can take up a lot of extra horizontal space and reduce readability. The leading @ in the line

```
var jobdir = @"\RedCentre\Jobs\Demo";
```

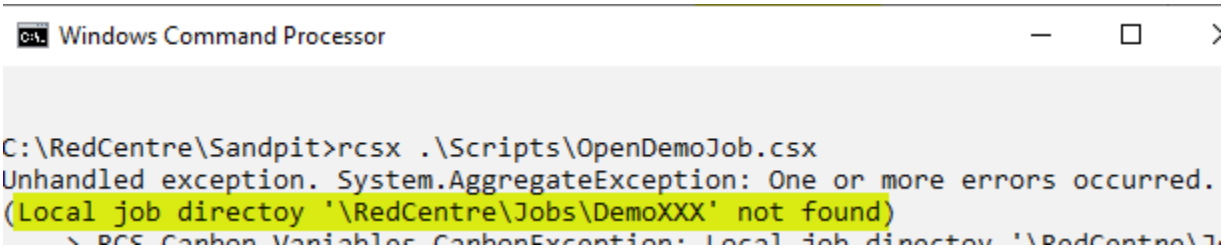
tells the compiler that \ is really \\. C languages use double forward slash // to indicate a comment or to disable lines.

The key line is *eng.OpenJob(jobdir).* This creates the job object inside the engine.

To test for error states, open this script in a text editor and add some extra characters to your ID, password or jobdir - for example, change the job name to *DemoXXX* as

```
var jobdir = @"\RedCentre\Jobs\DemoXXX";
```

and save the file. Running OpenDemoJob.csx reports the error and reason:



Revert your edits and resave the script.

Common Code - #load

The licence check and job assignment are tedious to repeat for every script. To reduce repeated code, we can hold it in another script and then use the *#load* command. *#load* is similar to *#include* in other languages.

Startup.csx

The startup script holds common start-up code and sets some upfront table properties.

Your licence credentials will always be the same, but the current job directory will not, so the intended job path is passed to *SetJobDir()* from the calling script. This approach hides your login credentials from working scripts, which you may want to share with others. You set your credentials for all your scripts in your local Startup.csx.

```
// Declare and define global engine eng, dprops and sprops objects
// Use eng to check the licence credentials
// Define SetJobDir wrapper
// Set some default table properties

var eng = new CrossTabEngine();

// Authentication
```

```
var summary = await eng.LoginId("16499372", "C6H12O6");
Console.WriteLine($"User: {summary.Name}");
Console.WriteLine($"DLL: {eng.Version}");

// Assign job
bool SetJobDir(string jobdir)
{
    try
    {
          eng.OpenJob(jobdir);
    }
    catch (Exception e)
    {
          Console.WriteLine($"Could not open job at {jobdir}. {e.Message}");
          return false;
    }
    Console.WriteLine($"Job: {eng.JobName}");
    return true;
}

// Default table properties for this job
var sprops           = new XSpecProperties() {};
var dprops           = new XDisplayProperties() {};
dprops.Output.Format = XOutputFormat.TSV;
dprops.Cells.Frequencies.Visible    = true;
dprops.Cells.ColumnPercents.Visible = true;
dprops.Cells.RowPercents.Visible    = false;
```

Since the primary purpose of a cross tabulation system is to specify and generate tables, it is convenient to set your preferred initial table properties here.

This creates the table's *XSpecProperties* and *XDisplayProperties* objects with their default settings and referenced as *sprops* and *dprops*.

```
var sprops  = new XSpecProperties() {};
var dprops  = new XDisplayProperties() {};
```

The defaults are then changed to whatever you may want as your standard output, here tab-delimited (TSV), frequency counts and column percents:

```
dprops.Output.Format = XOutputFormat.TSV;
dprops.Cells.Frequencies.Visible    = true;
dprops.Cells.ColumnPercents.Visible = true;
dprops.Cells.RowPercents.Visible    = false;
```

Like *eng*, *sprops* and *dprops* are global and can be accessed from any script which #loads a Startup.csx.

The specification and display settings can be changed at any time. See Appendix 3 for a list of all settings.

SetJobDir() in Startup.csx is a wrapper for *eng.OpenJob().*

Load this near the top of any script, and then call SetJobDir(<job directory>), which immediately returns if the job could not be opened. The ! character is C# for *not*.

```
#load "startup.csx"
if (!SetJobDir(@"C:\RedCentre\Jobs\Demo")) return;
```

If you enter a bad job path, the script will abort with an error message.

The startup script can be anywhere – usually in a parent directory to all the scripts which will #load it. For these examples, it is in \RedCentre\Sandpit, with path from the calling script in \Sandpit\Scripts\ as *..\startup.csx*.

RedCentreLibrary.csx

RedCentreLibrary.csx similarly wraps many common table and variable methods. As with *#load Startup.csx*, *#load RedCentreLibrary.csx* makes scripts shorter and easier to follow. The library file is stored in \RedCentre\Apps, as a sibling to the DLLs, because it can be used for any job. See *OneTable_RCSLib.csx* below for the first example. For convenience here, all example scripts which use the library are named **_RCSLib.csx*, and the non-library versons just *.csx. It is recommended that the RedCentre library is used for operations scripts.

TABLES

Running tables is done with the *GenTab* method, using much the same syntax for top, side, filter and weight as Ruby scripting. The spec and display properties (sprops and dprops) are strongly typed, meaning they are objects that need to be created. This is done in *Startup.csx*, as detailed above. Spec properties are settings which must be assigned before table generation (such as *InitAsMissing* or a case filter) and display properties are applied to the table post-generation.

OneTable.csx

```
#load "..\startup.csx"
if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;

var ret = eng.GenTab("test", "gender(cwf;*)", "region(cwf;*)", "", "", sprops, dprops);
Console.WriteLine(ret);
```

Windows Command Processor

```
C:\RedCentre\Sandpit>rcsx .\Scripts\OneTable.csx
User: CarbonGuest
DLL: 8.2.9.0
Job: Demo
Name: test
Top: Respondent Gender
Side: region
        Cases WF        Male    Female
Cases WF        10000   4985    5015
NE      2522    1255    1267
                25.18%  25.26%
SE      2459    1213    1246
                24.33%  24.85%
SW      2475    1230    1245
                24.67%  24.83%
NW      2544    1287    1257
                25.82%  25.06%
```

Global instances of the spec and display properties are declared and defined in *Startup.csx* as *sprops* and *dprops*.

The return from *GenTab* is string, but since the return type can be inferred by the compiler from eng.GenTab you can use just *var* instead, as

```
var ret = eng.GenTab(name,top,side,filter,weight,sprops,dprops);
```

Console.WriteLine displays the string.

```
Console.WriteLine(ret));
```

The output format (specified in Startup.csx) is tab-delimited.

OneTable_RCSLib.csx

The second line below loads the RedCentre library, which has a wrapper for *eng.GenTab*.

The GenTab wrapper simplifies the syntax. The *eng* object is hidden and since sprops and dprops are global, they do not need to be passed as the sixth and seventh parameters. This gives

```
#load "..\Startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"

if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;

var ret = GenTab("test", "gender(cwf;*)", "region(cwf;*)", "", "");
Console.WriteLine(ret);
```

The GenTab wrapper in RedCentreLibrary.csx is

```
public string GenTab(string name, string top, string side, string filt, string wght)
{
    try
    {
            return eng.GenTab(name, top, side, filt, wght, sprops, dprops);
    }
    catch (Exception e)
    {
            Console.WriteLine($"{e.Message} {OutputJobMessage()}");
            return null;
    }
}
```

This script also demonstrates the boilerplate for general operations work. All standard operational scripts should commence as

```
#load <path to your Startup.csx>
#load <path to \Apps\RedCentreLibrary.csx>
if (!SetJobDir(<your job path>)) return;
...
```

Writing to a File

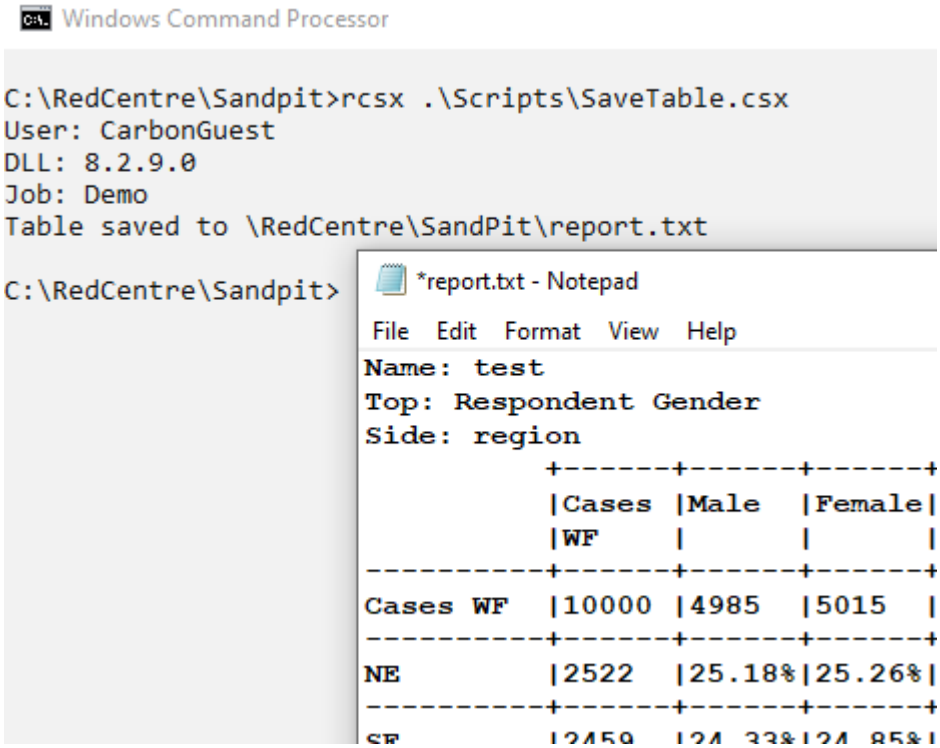
You can use the system File object for writing to a file.

SaveTable.csx

This script turns off frequencies (leaving just column% on for category cells), changes the output to formatted space (.SSV), appends the table output to a file, and then opens the file in Notepad.exe using the system method Process.Start().

```
#load "..\startup.csx"

if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;
var outfile = @"\RedCentre\SandPit\report.txt";

dprops.Cells.Frequencies.Visible = false;  // change a display property
dprops.Output.Format = XOutputFormat.SSV;  // space-aligned
var ret = eng.GenTab("test", "gender(cwf;*)", "region(cwf;*)", "", "", sprops, dprops);
```

```
File.AppendAllText(outfile, ret, Encoding.UTF8);  // append to existing if present
Console.WriteLine($"Table saved to {outfile}");
Process.Start("Notepad", outfile);
```



SaveTable_RCSLib.csx

The library version has the indicated changes:

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"

if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;
var outfile = @"\RedCentre\SandPit\report.txt";

dprops.Cells.Frequencies.Visible = false;  // change a display property
dprops.Output.Format = XOutputFormat.SSV;  // space-aligned
var ret = GenTab("test", "gender(cwf;*)", "region(cwf;*)", "", "");

File.AppendAllText(outfile, ret + "\n", Encoding.UTF8); // append to existing
Console.WriteLine($"Table saved to {outfile}");
OpenTextFile(outfile);
```

OpenTextFile is defined in RedCentreLibrary.csx as

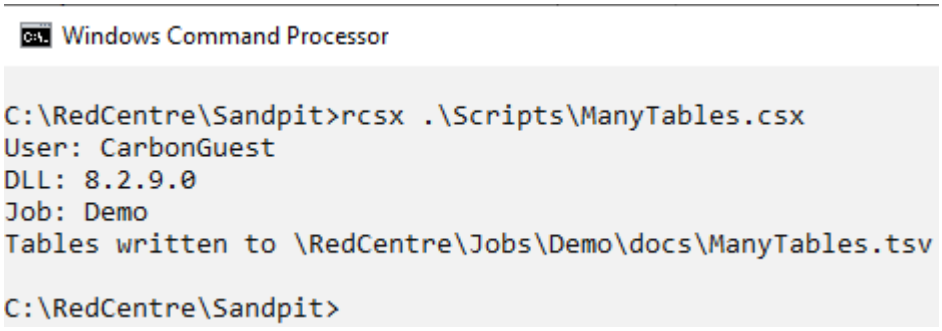
```
public void OpenTextFile(string fname)
{
    Process.Start("NotePad", fname);
}
```

Output Many Tables

You can run many tables and save to a single file using the internal *OutputManager* class. A typical sequence is:

ManyTables.csx

```
#load "..\startup.csx"
if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;

eng.OutputManager.Open(XOutputFormat.TSV, @"docs\ManyTables.tsv");
eng.OutputManager.AppendLine("\n   *** Gender by Region ***\n");   // \n is new line
eng.GenTab("tab1", "gender", "region", "", "", sprops, dprops);
eng.OutputManager.AppendTable();
eng.OutputManager.AppendLine("\n   *** Age by Education ***\n");
eng.GenTab("tab2", "age", "edu", "", "", sprops, dprops);
eng.OutputManager.AppendTable();
eng.OutputManager.Close();
Console.WriteLine("Tables written to " + eng.OutputManager.Message);
```



Note that the *dprops.Output.Format* setting in Startup.csx will be overridden. You set the format with *eng.OutputManager.Open* and the output manager will handle it from there.

If you supply a job-relative filename for the destination, output goes there - otherwise it is in *eng.OutputManager.Message().*

The basic sequence is to generate then output each table, interspersing output lines for some spacing, sub-headings or commentary.

AppendLine takes a string. If can be one line or many lines separated by \n.

If anything goes wrong, the output commands throw exceptions that will be reported as errors in the console window.

Supported formats are TSV, CSV, SSV, XML, HTML and XLSX. Browsers may have trouble with large HTML files, so be careful. The practical limit seems to be about 4,000 lines. HTML can also be opened in Excel. SSV is a basic space-delimited grid output.

ManyTables_RCSLib.csx

There are wrappers for these methods which hide the engine and output manager objects. If you *#load RedCentreLibrary.csx* then the above can be rewritten without the leading *eng.OutputManager,* using the 5 parameter GenTab wrapper, and the short forms for open, output table, output line, close and output message as

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"

if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;

OpenOutput(XOutputFormat.TSV, @"docs\ManyTables.tsv");
OutputLine("\n   *** Gender by Region ***\n");   // \n is new line
GenTab("tab1", "gender", "region", "", "");
OutputTable();
```

```
OutputLine("\n   *** Age by Education ***\n");
GenTab("tab2", "age%", "edu", "", "");
OutputTable();
CloseOutput();
Console.WriteLine("Tables written to " + OutputMessage());
```

THE DEFCON FAMILY

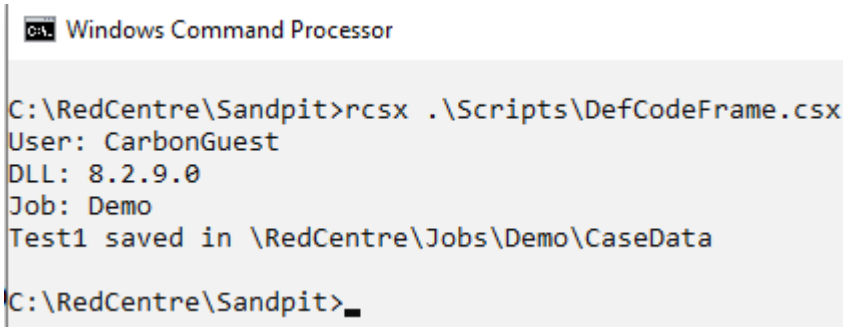
There is a set of objects that reproduce the *DefCodeFrame* and *DefCon/DefGen* functions in the Ruby libraries for creating variables. As much as possible the syntax and method names have been preserved so converting Ruby scripts to Carbon CSX requires minimal change. See Appendix 4 for the current *DefCon* coverage and how to convert from Ruby VB script to Carbon CSX script.

DefCodeFrame

Use the *DefCodeFrame* object to create codeframes (*.MET files). This example creates the BrandX codes directly, the BrandY codes in a loop, and the BrandZ codes by adding multiple items in a single line. Nets for each brand are also added as a single line.

DefCodeFrame.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

var dcf = eng.NewDefCodeFrame("Test1", "Example DefCodeFrame");
if (dcf==null)
{
   Console.WriteLine(eng.Message());
   return;
}

//BrandX
dcf.AddCode(1, "Brand1");
dcf.AddCode(2, "Brand2");
dcf.AddCode(3, "Brand3");

//BrandY
for(int i = 4; i<= 6; i++)
    dcf.AddCode(i, $"Brand{i}");

//BrandZ
dcf.AddItems("7=Brand7\n8=Brand8\n9=Brand9\n10=Brand10=BrandY");

//Nets
dcf.AddItems("_net(1/3)=BrandX\n_net(4/6)=BrandZ\n_net(7/10)=BrandZ");

if (!dcf.Close()) Console.WriteLine(eng.Message());
else              Console.WriteLine(@$"Test1 saved in {jobdir}\CaseData");
```



You create a new *DefCodeFrame* object, here referenced as *dcf,* passing the variable name (here *Test1*) and description, and then call *AddCode* and other methods to build the codeframe.

This shows C# loop syntax and the neat way to assemble strings using $ and {}. The *DefCodeFrame* methods are *AddCode(), AddArith(), AddNet(), AddExpr(), AddItems(), RemoveItems(), Copy()* and *Close()*. See Appendix 4 for details.

Resultant MET file is

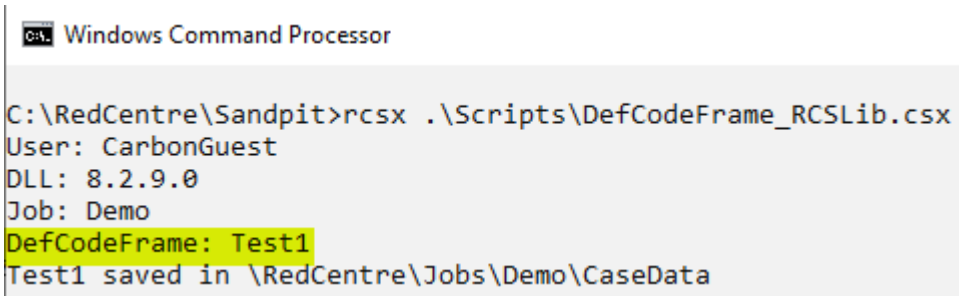
```
[Codeframes]
(Test1)
1=Code1 j1
2=Code2 j3
3=Code3 j5
4=Code4 j7
5=Code5 j9
()

[MetaData]
Desc=Example DefCodeFrame
Base0=cwf
DataMJD=0
Cases=0
```

DefCodeFrame_RCSLib.csx

Codeframe specifications can be simplified by using the library wrappers.

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

DefCodeFrame("Test1", "Example DefCodeFrame");

//BrandX
AddCode(1, "Brand1");
AddCode(2, "Brand2");
AddCode(3, "Brand3");

//BrandY
for(int i = 4; i<= 6; i++)
    AddCode(i, $"Brand{i}");

//BrandZ
AddItems("7=Brand7\n8=Brand8\n9=Brand9\n10=Brand10=BrandY");

//Nets
AddItems("_net(1/3)=BrandX\n_net(4/6)=BrandZ\n_net(7/10)=BrandZ");

CodeFrameClose();
Console.WriteLine(@$"Test1 saved in {jobdir}\CaseData");
```

```
OpenTextFile(@$"{jobdir}\CaseData\Test1.met");
```



Note that the library version also outputs a progress message. Many library wrappers output progress messages.

DefCon

You create a DefCon object, here referenced as *dcn*, passing the variable name and description, and then call *AddItem()* and other routines to build the construction.

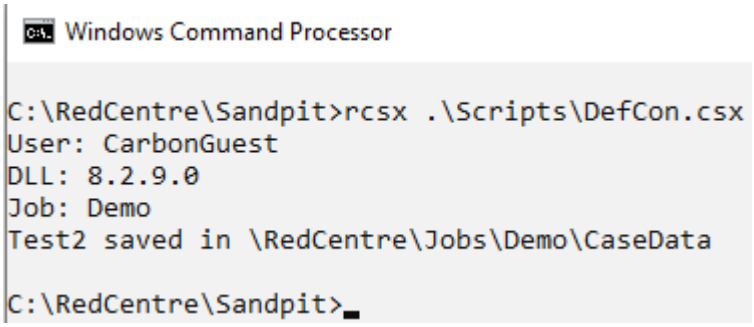
This just writes the MET file, which you could construct with *eng.Construct(varname)* or by passing *true* in the Close call.

```
dcn.Close(true);
```

The *DefCon* subroutines are *AddItem(), AddInc(), AddNextItem(), AddExpr(), SetFlags()* and *Close().* See Appendix 4 below for details.

DefCon.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

var dcn = eng.NewDefCon("Test2", "Example DefCon");
if (dcn==null)
{
   Console.WriteLine(eng.Message());
   return;
}
dcn.AddItem(1, "Age(1/2)", "15yrs to 35yrs");
dcn.AddItem(2, "Age(3/4)", "36yrs to 65yrs");
dcn.AddItem(3, "Age(5)",   "65yrs and above");
dcn.Close(true);

Console.WriteLine(@$"Test2 saved in {jobdir}\CaseData");
```



The resultant MET file (\RedCentre\Jobs\Demo\CaseData\Test2.met) is

```
[Codeframes]
(Test2)
1=15yrs to 35yrs
2=36yrs to 65yrs
3=65yrs and above
()

[Constructor]
Flags=CI
1=Age(1/2)
2=Age(3/4)
3=Age(5)

[MetaData]
Desc=Example DefCon
Base0=cwf
DataMJD=0
Cases=10000
```

DefCon_RCSLib.csx

DefCon specifications can be simplified by using the library wrappers.

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

DefCon("Test2", "Example DefCon");
  AddItem(1, "Age(1/2)", "15yrs to 35yrs");
  AddItem(2, "Age(3/4)", "36yrs to 65yrs");
  AddItem(3, "Age(5)",   "65yrs and above");
ConClose(true);

Console.WriteLine(@$"Test2 saved in {jobdir}\CaseData");
```

DefGen

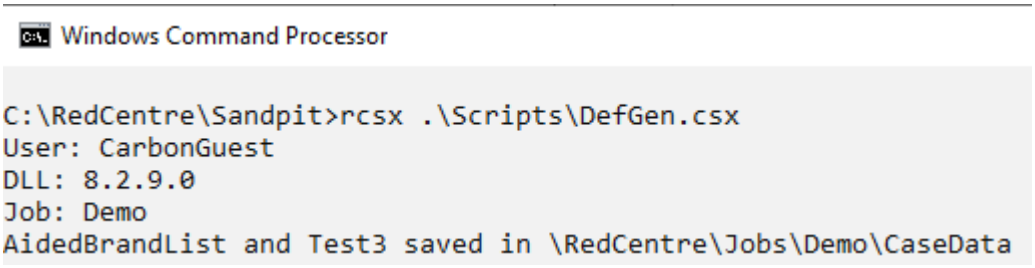
Use the *DefGen* object to generate the metadata (*.met) specification for constructed variables.

First, we create a codeframe called *AidedBrandList*, then the generator uses the new code frame as the outer level (or top axis) of a Net Promoter Score (NPS) ratings grid.

DefGen.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

var dcf = eng.NewDefCodeFrame("AidedBrandList", "");
if (dcf==null)
{
   Console.WriteLine(eng.Message());
   return;
}
dcf.AddCode(1, "BrandX");
dcf.AddCode(2, "BrandY");
dcf.AddCode(3, "BrandZ");
dcf.Close();

var dgn = eng.NewDefGen("Test3", "NPS_$a($b)", "", "Example DefGen");
if (dgn==null)
{
   Console.WriteLine(eng.Message());
   return;
```

```
}
dgn.AddLevel("a", "Brand",  "AidedBrandList");
dgn.AddLevel("b", "Rating", "NPS_1");
dgn.Close(true);

Console.WriteLine(@$"AidedBrandList and Test3 saved in {jobdir}\CaseData");
```



You create a new *DefGen* object, here referenced as *dgn*, passing the variable name, the filter generation expression, the increment expression (here "") and the description; then call *AddLevel* and other routines to specify the construction. *Close(true)* generates and constructs the variable immediately.

Other commands are *AddLevel(), SetFlags()* and *Close().* See Appendix 4 for details.

Resultant generated MET file Test3.met is

```
[Codeframes]
(Brand)
1=BrandX
2=BrandY
3=BrandZ
()

(Rating)
1=1
2=2
3=3
4=4
5=5
6=6
7=7
8=8
9=9
10=10
()

[Constructor]
Flags=CI
1:1=NPS_1(1)
1:2=NPS_1(2)
1:3=NPS_1(3)
1:4=NPS_1(4)
1:5=NPS_1(5)
1:6=NPS_1(6)
1:7=NPS_1(7)
1:8=NPS_1(8)
1:9=NPS_1(9)
1:10=NPS_1(10)
2:1=NPS_2(1)
2:2=NPS_2(2)
2:3=NPS_2(3)
2:4=NPS_2(4)
2:5=NPS_2(5)
2:6=NPS_2(6)
2:7=NPS_2(7)
```

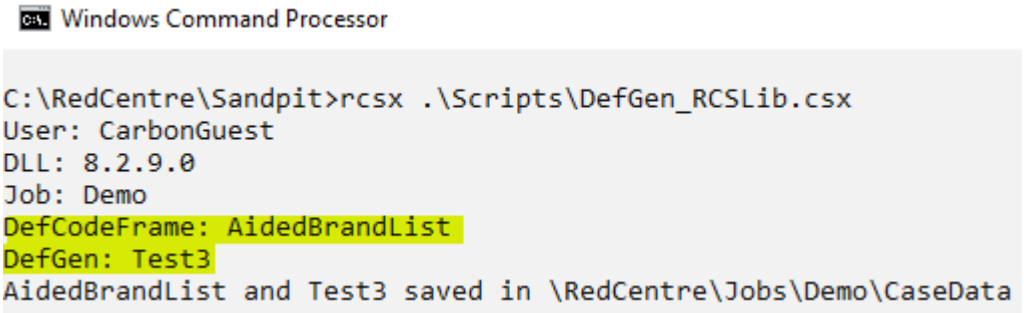
```
2:8=NPS_2(8)
2:9=NPS_2(9)
2:10=NPS_2(10)
3:1=NPS_3(1)
3:2=NPS_3(2)
3:3=NPS_3(3)
3:4=NPS_3(4)
3:5=NPS_3(5)
3:6=NPS_3(6)
3:7=NPS_3(7)
3:8=NPS_3(8)
3:9=NPS_3(9)
3:10=NPS_3(10)

[Generator]
Filter=NPS_$a($b)
Increment
Flags
a=nBrand sAidedBrandList r* l$AidedBrandList($)
b=nRating sNPS_1 r* l$NPS_1($)

[MetaData]
Desc=Example DefGen
Base0=cwf
Base1=cwf
DataMJD=0
Cases=10000
```

DefGen_RCSLib.csx

DefGen specifications can be simplified by using the library wrappers.

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

DefCodeFrame("AidedBrandList", "");
   AddCode(1, "BrandX");
   AddCode(2, "BrandY");
   AddCode(3, "BrandZ");
CodeFrameClose();

DefGen("Test3", "NPS_$a($b)", "", "Example DefGen");
   AddLevel("a", "Brand",  "AidedBrandList");
   AddLevel("b", "Rating", "NPS_1");
GenClose(true);

Console.WriteLine(@$"AidedBrandList and Test3 saved in {jobdir}\CaseData");
```

Note the extra progress indicators.



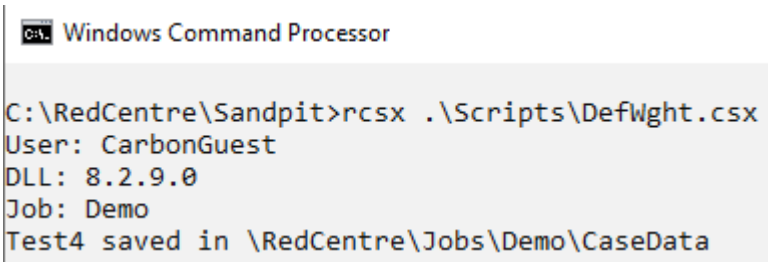
This is essential for long and complicated scripts.

DefWght

Use the *DefWght* object to create weight variables.

DefWght.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

var dwt = eng.NewDefWght("Test4", "Cell", "Month", "", 0, 0, "Example DefWght");
if (dwt==null)
{
   Console.WriteLine(eng.Message());
   return;
}
dwt.AddTarget("Age",    "10;30;30;20;10");
dwt.AddTarget("Gender", "40;60");
dwt.AddTarget("Region", "25;25;25;25");
dwt.Close(true);

Console.WriteLine(@$"Test4 saved in {jobdir}\CaseData");
```



You create a *DefWght* object, here referenced as *dwt*, passing the variable name, type (Cell/Rim), period variable, filter expression, rim difference, target population and the description; then call *AddTarget* to build the construction. Note *Close()* can be *Close(true)* to generate and construct the variable immediately.

Resultant MET Test4.met is

```
[Special]
Type=Weight.Cell
Ancestors=Age,Gender,Region,Month
Period=Month
(Variables)
Age=10;30;30;20;10
Gender=40;60
Region=25;25;25;25
()

[MetaData]
Desc=Example DefWght
DataMJD=0
Cases=10000
```

DefWght_RCSLib.csx

DefWght specifications can be simplified by using the library wrappers.

```
#load "..\startup.csx"
```

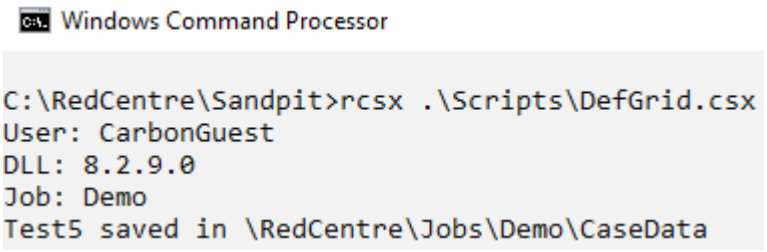
```
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

DefWght("Test4", "Cell", "Month", "", 0, 0, "Example DefWght");
  AddTarget("Age",    "10;30;30;20;10");
  AddTarget("Gender", "40;60");
  AddTarget("Region", "25;25;25;25");
WghtClose(true);

Console.WriteLine(@$"Test4 saved in {jobdir}\CaseData");
```

DefGrid

Use the *DefGrid* object to create grid variables. This is an alternative to using the more general *DefGen* and does not require intermediate codeframes.

DefGrid.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

var dgr = eng.NewDefGrid("Test5","copy of NPS", "Brand", "NPS_1;NPS_2;NPS_3", "Score",
"NPS_1");
if (dgr==null)
{
   Console.WriteLine(eng.Message());
   return;
}
dgr.Close(true);

Console.WriteLine(@$"Test5 saved in {jobdir}\CaseData");
```



You create a *DefGrid* object, here referenced as *dgr*, passing the variable name, description, the top name, top codeframe(s), side name and side codeframe(s). *Close(true)* generates and constructs the variable immediately.

The resultant MET file *Test5.met* is

```
[Codeframes]
(Level1)
1=NPS_1
2=NPS_2
3=NPS_3
()

(Level2)
1=1
2=2
3=3
4=4
5=5
```

```
6=6
7=7
8=8
9=9
10=10
()

[Constructor]
Flags=CI
1:1=NPS_1(1)
1:2=NPS_1(2)
1:3=NPS_1(3)
1:4=NPS_1(4)
1:5=NPS_1(5)
1:6=NPS_1(6)
1:7=NPS_1(7)
1:8=NPS_1(8)
1:9=NPS_1(9)
1:10=NPS_1(10)
2:1=NPS_2(1)
2:2=NPS_2(2)
2:3=NPS_2(3)
2:4=NPS_2(4)
2:5=NPS_2(5)
2:6=NPS_2(6)
2:7=NPS_2(7)
2:8=NPS_2(8)
2:9=NPS_2(9)
2:10=NPS_2(10)
3:1=NPS_3(1)
3:2=NPS_3(2)
3:3=NPS_3(3)
3:4=NPS_3(4)
3:5=NPS_3(5)
3:6=NPS_3(6)
3:7=NPS_3(7)
3:8=NPS_3(8)
3:9=NPS_3(9)
3:10=NPS_3(10)

[Grid]
Size=10x3
TopFrames=NPS_1;NPS_2;NPS_3
SideFrame=NPS_1

[MetaData]
Desc=copy of NPS
Base0=cwf
Base1=cwf
DataMJD=0
Cases=10000
```

To invert the grid (rating score on top, brands on the side), change the arguments as

```
var dgr = eng.NewDefGrid("Test5","copy of NPS", "Score", "NPS_1", "Brand",
"NPS_1;NPS_2;NPS_3");
```

DefGrid_RCSLib.csx

DefGrid specifications can be simplified by using the library wrappers.

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

DefGrid("Test5","copy of NPS", "Brand", "NPS_1;NPS_2;NPS_3", "Score", "NPS_1");
GridClose(true);
```

```
Console.WriteLine(@$"Test5 saved in {jobdir}\CaseData");
```

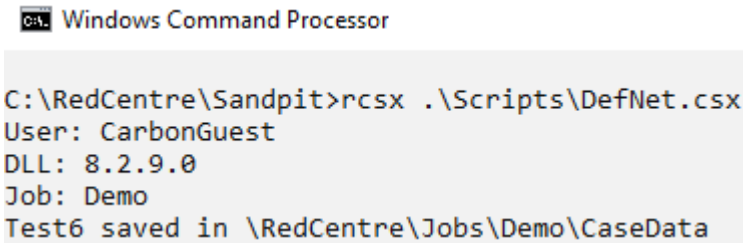
DefNet

Use the *DefNet* object to create an arbitrary set of nets at any hierarchic level.

DefNet.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

var dn = eng.NewDefNet("Test6", "Example DefNet");
  var dnl = dn.AddLevel("Region");
      dnl.AddItem("region(1/2)",  "North");
      dnl.AddItem("region(3/4)",  "South");
  dnl = dn.AddLevel("Work");
      dnl.AddItem("occupation(1/2)", "Professional");
      dnl.AddItem("occupation(2/3)", "Midlevel");
      dnl.AddItem("occupation(4/5)", "Other");
dn.Close(true);

Console.WriteLine(@$"Test6 saved in {jobdir}\CaseData");
```

Note this example has no error checking. The library version below does error checking for you.



You create a *DefNet* object, here referenced as *dn*, passing the variable name and description. From there you create levels with items underneath them. Each item is a filter expression and label. The result is a hierarchic variable built from the filters at each level. *Close(true)* generates and constructs the variable immediately.

The resultant MET file Test6.met is

```
[Codeframes]
(Region)
1=North
2=South
()

(Work)
1=Professional
2=Midlevel
3=Other
()

[Constructor]
Flags=CI
1:1=region(1/2)&occupation(1/2)
1:2=region(1/2)&occupation(2/3)
1:3=region(1/2)&occupation(4/5)
2:1=region(3/4)&occupation(1/2)
2:2=region(3/4)&occupation(2/3)
2:3=region(3/4)&occupation(4/5)
```

```
[MetaData]
Desc=Example DefNet
Base0=cwf
Base1=cwf
DataMJD=0
Cases=10000
```

DefNet_RCSLib.csx

DefNet specifications can be simplified by using the library wrappers.

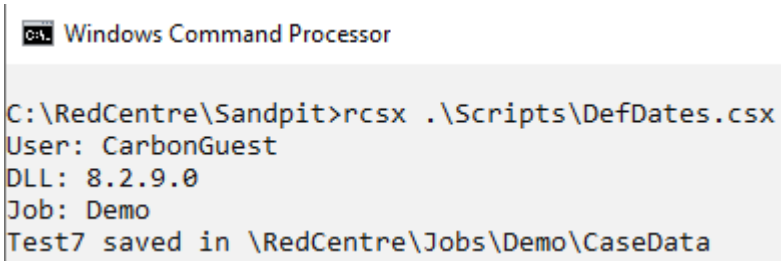
```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

DefNet("Test6", "Example DefNet");
  AddNetLevel("Region");
    AddNetItem("region(1/2)",  "North");
    AddNetItem("region(3/4)",  "South");
  AddNetLevel("Work");
    AddNetItem("occupation(1/2)", "Professional");
    AddNetItem("occupation(2/3)", "Midlevel");
    AddNetItem("occupation(4/5)", "Other");
NetClose(true);

Console.WriteLine(@$"Test6 saved in {jobdir}\CaseData");
```

DefDates

Use the DefDates object to create date variables as Modified Julian Dates (the day number since 17Nov1858).

DefDates.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

var ddt = eng.NewDefDates("Test7", "Date", "1y 1Jan2001", "Day",  "mmmyyyy", "Example
DefDates");
if (ddt==null)
{
   Console.WriteLine(eng.Message());
   return;
}
ddt.Close(true);

Console.WriteLine(@$"Test7 saved in {jobdir}\CaseData");
```



Everything is in the parameters passed in the definition: name, type, cycle, source, format and description. *Close(true)* generates and constructs the variable immediately.

The resultant MET file Test7.met is

```
[Codeframes]
(Test7)
1=Jan2021
2=Jan2022
3=Jan2023
4=Jan2024
()

[Constructor]
Flags=CI
1=Day(1/255)
2=Day(256/510)
3=Day(511/765)
4=Day(766/1000)

[DateGen]
Method=Date
Cycle=1y 1Jan2001
Source=Day
SourceFormat=
Format=mmmyyyy

[MetaData]
Desc=Example DefDates
Base0=cwf
DataMJD=1
Dates=1y 1Jan2001
Cases=10000
```

DefDates_RCSLib.csx

DefDate specifications can be simplified by using the library wrappers.

```
#r "\RedCentre\Apps\RCS.Carbon.Tables.dll"
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Jobs\Demo";
if (!SetJobDir(jobdir)) return;

DefDates("Test7", "Date", "1y 1Jan2001", "Day",  "mmmyyyy", "Example DefDates");
DatesClose(true);

Console.WriteLine(@$"Test7 saved in {jobdir}\CaseData");
```

Title Text Modes

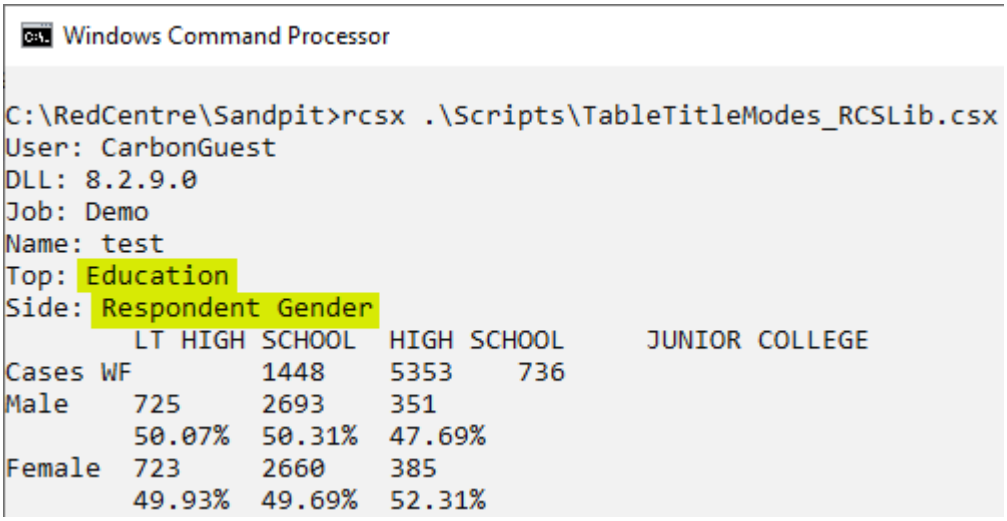
The Top and Side title text can be auto-generated according to four display property settings.

TableTitleModes.csx

```
#load "..\Startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"

if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;

//dprops.Titles.Labelling.Script = true;
//dprops.Titles.Labelling.Codes = true;
//dprops.Titles.Labelling.Name = true;
//dprops.Titles.Labelling.Desc = true;
```

```
var ret = GenTab("test", "EDU(1/3)", "Gender(cwf;*)", "", "");
Console.WriteLine(ret);
```

Note the four commented lines. The default behaviour is to show the variable descriptions if present.



Uncomment subsets of the four Labelling statements as indicated below, save and run. The top title should follow the text in blue.

.Desc=true (or all Labelling properties false)
Top: Education
Just variable description.

.Script=true
Top: edu(1/3)
No parsing at all, just script from the spec (cannot be mixed with Codes, Desc or Names)

.Codes=true, .Desc=true
Top: Education(LT HIGH SCHOOL to JUNIOR COLLEGE)
Description/Label for var and codes

.Codes=true,.Name=true
Top: edu(1/3)
Ends up the same as Script but actually goes through the parsing routine

.Name=true,.Desc=true
Top: edu - Education
Name and desc for var but no codes

.Codes=true,.Name=true,.Desc=true
Top: edu - Education(1-LT HIGH SCHOOL to 3-JUNIOR COLLEGE)
All in

Title, Variable and Code Label Overrides

If the table titles or row/column labels as delivered by the source or constructed labelling regimes is inappropriate, any label item can be overridden by following the specification part with \...\.

This is done according to these rules:

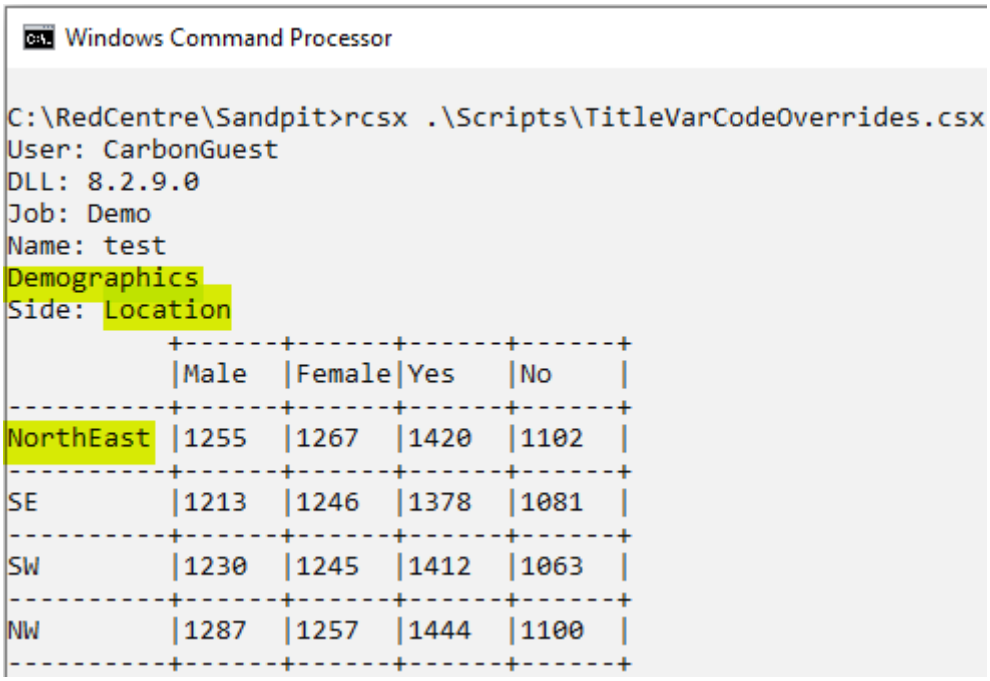
1. varname\variable description override\ to show the description in the title, retaining the Top: or Side: prefix
2. varname,varname,…varname,\title override\ to replace the title item completely
3. varname(1\label for code 1\;…) to replace a code label

Note the comma before the \ in rule#2 – otherwise the override would apply to the last variable only.

TitleVarCodeOverrides.csx

```
#load "..\startup.csx"
if (!SetJobDir(@"\RedCentre\Jobs\Demo")) return;

dprops.Output.Format = XOutputFormat.SSV;
dprops.Cells.ColumnPercents.Visible = false;

var ret = eng.GenTab("test", @"gender,married,\Demographics\",
                     @"region(1\NorthEast\;2/4)\Location\", "", "", sprops, dprops);
Console.WriteLine(String.Join(Environment.NewLine, ret));
```

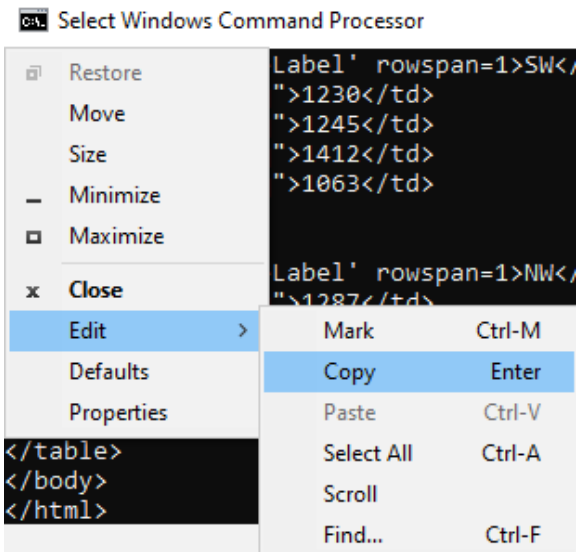
If there are a lot of overrides then use a leading @ to avoid multiple \\.



If the output format type supports group labels (types HTML and XLSX) then overridden variable text will show in the groups. Change the output format to

```
dprops.Output.Format = XOutputFormat.HTML;
```

save, rerun, then select and copy/paste the console output to Excel:



	A	B	C	D	E	F
1	Name: test					
2	Demographics					
3	Side: Location					
4	Key Frequencies		Respondent Gender		married	
5			Male	Female	Yes	No
6		NorthEast	1255	1267	1420	1102
7	Location	SE	1213	1246	1378	1081
8		SW	1230	1245	1412	1063
9		NW	1287	1257	1444	1100

Change *.HTML* back to *.SSV* and resave.

The Gender variable has the description *Respondent Gender* defined at the source level, so that is what is shown as the group label. You will generally want to override the title items so that the tables are well described for analysis, but all variables, unless obvious from the name, should have descriptions defined by either the case data source metadata or by your construction regime. You should only occasionally need to override variable descriptions or code labels for brevity and/or clarity.

GENERATING CONSTRUCTIONS

Manual Build

DefCon adds construction lines that are written to the MET file with *Close(). DefWght* writes the weighting matrix specification to the MET file. You can explicitly run those constructions to create the CD data file with

```
eng.Construct(varname);
```

DefGen and *DefGrid* need to generate their construction lines in preparation for a run, so the sequence is:

```
eng.Generate(varname);
eng.Construct(varname);
```

You would delay the generation and construction steps if you wanted to create all the specifications up-front and then run them all together at a later processing step. You may want to check things before committing, especially if the job is huge and constructing will take a long time.

Auto Build

Alternatively, you can trigger immediate construction as part of the *Close()* call, as

```
dg.Close(true);
```

The *Close* call takes an optional parameter to generate and construct the variable on the spot.

Note : *DefCodeFrame.Close()* does not take *true/false* because it only writes the MET file – no generating or constructing is performed.

Error Messages

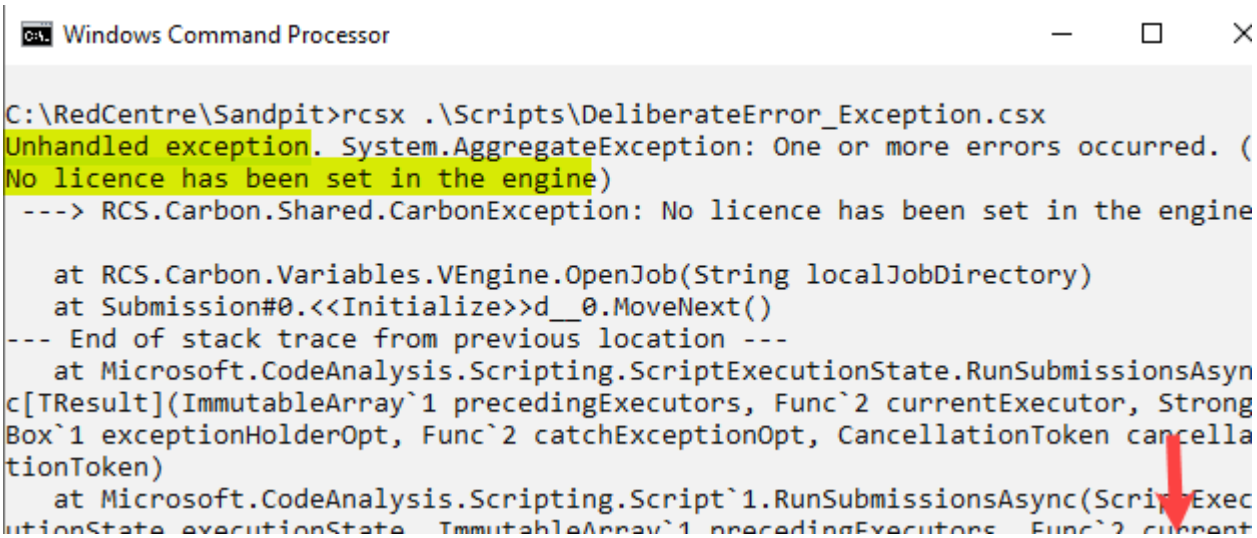
If things go wrong, there are two types of messages.

Exceptions

Run the script DeliberateError_Exception.csx:

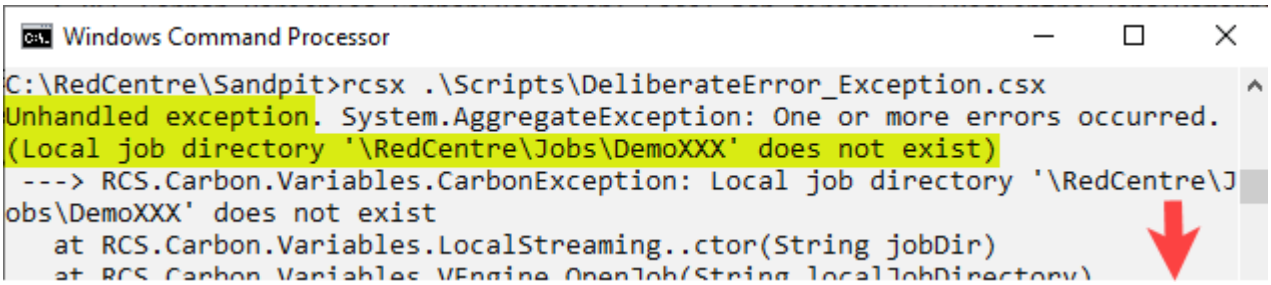
```
var eng = new CrossTabEngine();
//var summary = await eng.LoginId("16499372", "C6H12O6");
eng.OpenJob(@"\RedCentre\Jobs\DemoXXX");
```

The obvious error here is that there is no login (it has been commented out).



Exceptions are fatal, so the script aborts and the system shows as much information about the error as can be obtained. Uncomment the login step, save and rerun.

```
var eng = new CrossTabEngine();
var summary = await eng.LoginId("16499372", "C6H12O6");
eng.OpenJob(@"\RedCentre\Jobs\DemoXXX");
```

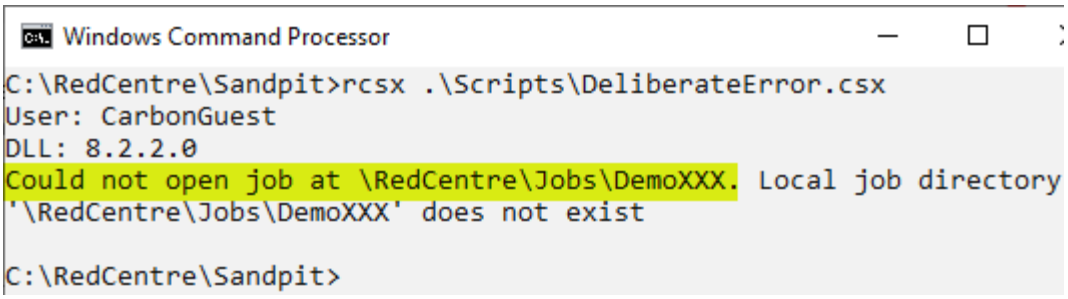


Again the cause is obvious – there is no such folder as \DemoXXX.

Now run the script DeliberateError.csx.

```
r "..\..\Apps\RCS.Carbon.Tables.dll"
#load "..\startup.csx"
SetJobDir(@"\RedCentre\Jobs\DemoXXX");
```

Most of the extra exception information is not useful, so if you #load Startup.csx and call SetJobDir(), the exception is handled for you and the error reported as simply



Messages

There is also a message string you can retrieve if things go wrong that do not throw an exception.

```
string msg = eng.Message();
```

You can ask for this at any time or if a call fails – for instance

```
if(!eng.Generate("mygrid")) Console.WriteLine(eng.Message());
```

The Generate() call returns true/false. If it fails (***!****eng.Generate()* means it returned false) you can send the message to the console.

This is good for the auto build lines where you are bundling *Generate()* and *Construct()* into a Close(true) call.

```
if(!dg.Close(true)) Console.WriteLine(eng.Message());
```

IMPORT

There are some preliminary imports covering file types .sav, .tsv and .csv.

The general call is <importengine>.ImportVars(file, settings*).*

Note that importing is not done using the crosstab engine. Imports (and exports) are handled by dedicated engines to enforce a separation of functionality.

You can only import into a new job (or overwrite an existing one) – there is no support for wave importing yet.

Create a job by making a subdirectory anywhere (this will be the name of the job).

This example uses the C:\RedCentre\Sandpit\TestImport\ directory.

The source file *Demo_dems_2021.sav* is already in the \TestImport\Source subdirectory.

An import test script is:

ImportDemoDems.csx

```
#load "..\startup.csx"
var jobdir = @"\RedCentre\Sandpit\TestImport";
```

```
if (!SetJobDir(jobdir)) return;

var sourcefile = "Demo_dems_2021.sav";  // relative to <jobdir>\Source\

var imp = new ImportEngine();
imp.AttachJob(eng);

ImportSavSettings settings = new ImportSavSettings();
settings.TryBlend = false;
Console.WriteLine("Importing: " + sourcefile);
var ret = imp.ImportSAV(sourcefile, settings);
Console.WriteLine(ret);
```

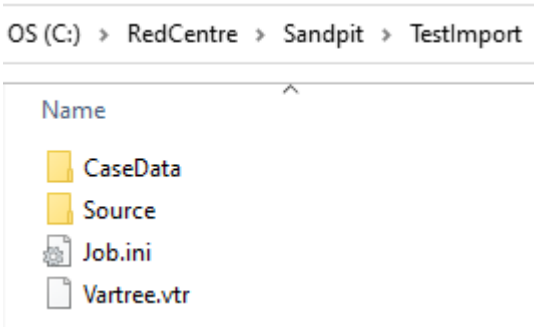
Note the *jobdir* assignment is different (not Demo anymore).

An RCS job is just a directory which could be anywhere, and you move/copy/rename a job by moving/copying/renaming the directory.

Create the import engine, referenced as *imp*, and attach the current engine job.

Create a settings object which sets various flags to control the import.

Supply the source file name as the first parameter (relative to \<job path>\Source), and the import settings as the second. The settings vary according to the import type.



The TestImport folder now looks like a normal RCS job with a Job.ini, a Vartree.vtr, and a CaseData folder.

ImportDemoDems_RCSLib.csx

The RCS Library version hides the import engine.

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Sandpit\TestImport";
if (!SetJobDir(jobdir)) return;

var sourcefile = "Demo_dems_2021.sav";  // relative to <jobdir>\Source\
ImportSavSettings settings = new ImportSavSettings();
settings.TryBlend = false;
ImportSAV(sourcefile, settings);
```

EXPORT

There are a few preliminary export types: CSV, TSV and Tableau.

The general call is <exportengine>.ExportVars(file, settings*).*

Note that exporting is not done with the local job engine. Exports (and imports) are handled by dedicated engines to enforce a separation of functionality.

The file is just a file name or stem – no path, and no extension, because exports could be two or more files (data and metadata) with different extensions inferred by the format type.

Output files are always written to a local directory. That can be a full path. So

```
new ExportSettings("d:\\1jobs\\output\\TestExport")
```

will write (for TSV) d:\\1jobs\\output\\TestExport.tsv, or a job relative path

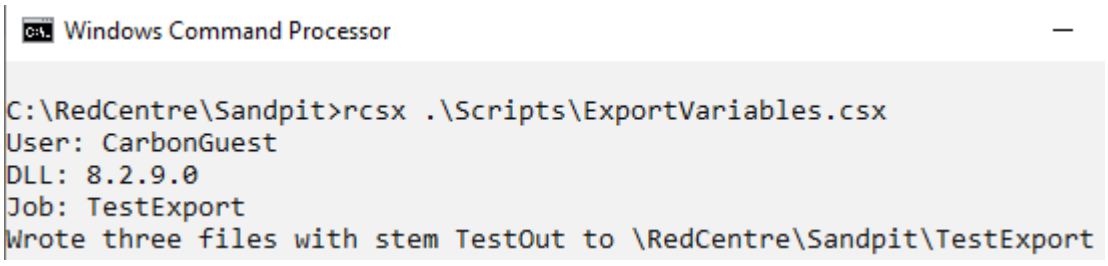
```
new ExportSettings("docs\\TestOut")
```

For a local job the path is relative to the current job directory. For an online job the output path is set by the call eng.SetHomeDir(path). For an example see the script \RedCentre\Sandpit\Scripts\ExportVars_Azure.csx.

ExportVariables.csx uses the previous import example, but with UBA, a multi-response variable, manually copied from the Demo job.

ExportVariables.csx

```
#load "..\startup.csx"

var jobdir = @"\RedCentre\Sandpit\TestExport";
if (!SetJobDir(jobdir)) return;

string exportvars = "Region,AgeX,UBA";
var exp = new ExportEngine();
exp.AttachJob(eng);
var settings = new ExportSettings("TestOut")
{
    Format = VExportType.Tableau
};
var result = exp.ExportVars(exportvars, settings);
Console.WriteLine(result);
```



Start by listing the variables to be exported as a comma-delimited string.

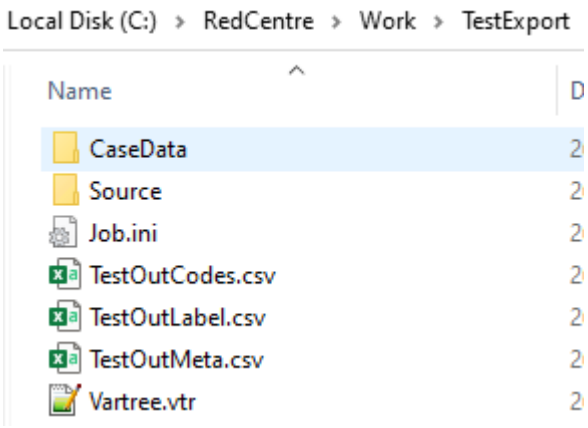
Create an *ExportSettings* object with the destination file stem and set any parameters in its constructor. You must set the Format to one of *EngineExportType.TSV*, *EngineExportType.CSV* or *EngineExportType.Tableau* – see Appendix 5. Other export types (following Ruby) will be provided later.

ExportVariables_RCSLib.csx

The RCS Library version eliminates several lines.

```
#load "..\startup.csx"
#load "..\..\Apps\RedCentreLibrary.csx"
var jobdir = @"\RedCentre\Sandpit\TestExport";
if (!SetJobDir(jobdir)) return;
```

```
string exportvars = "Region,AgeX,UBA";
var settings = new ExportSettings("TestOut")
{
    Format = VExportType.Tableau
};
ExportVars(exportvars, settings);
```



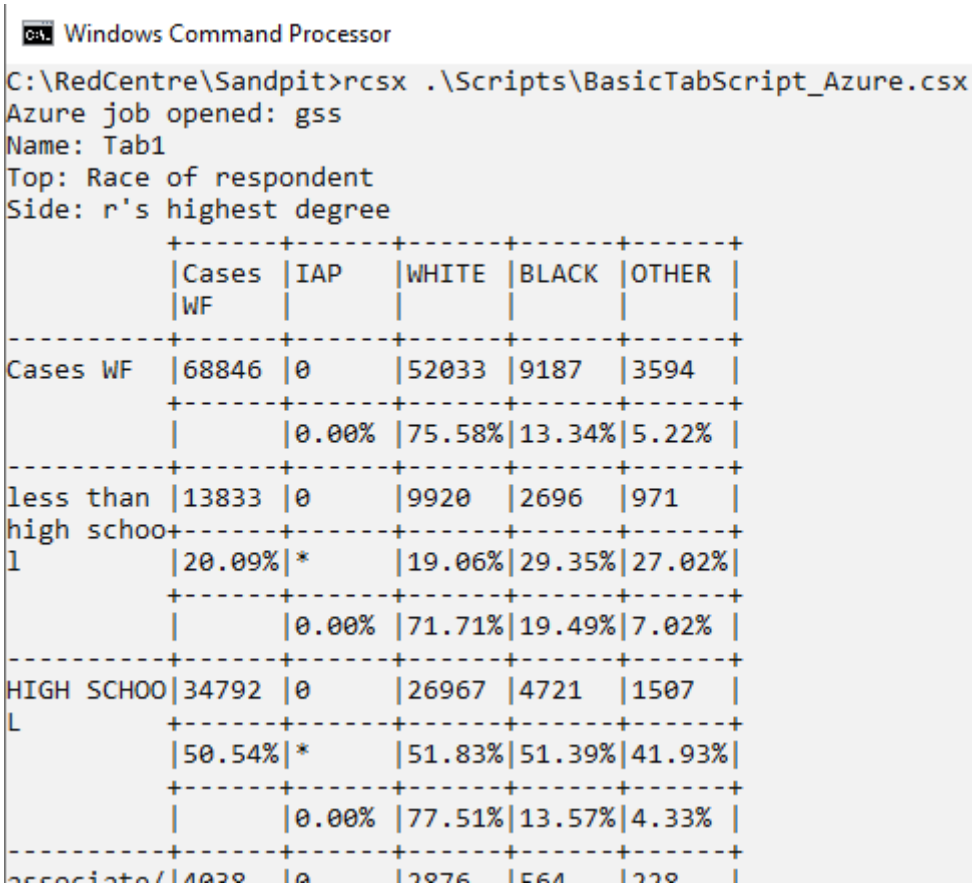
The Tableau export uses the destination file stem and writes three *.csv files which can be used as the inputs for a Tableau hypercube.

Accessing Cloud Jobs

The only difference for accessing a cloud job is the OpenJob command. If called with two parameters, as customer account and job name, then the Azure connection is automatically established, and all case data reads will be from the Azure storage. A stable and fast internet connection is advised.

BasicTabScript_Azure.csx

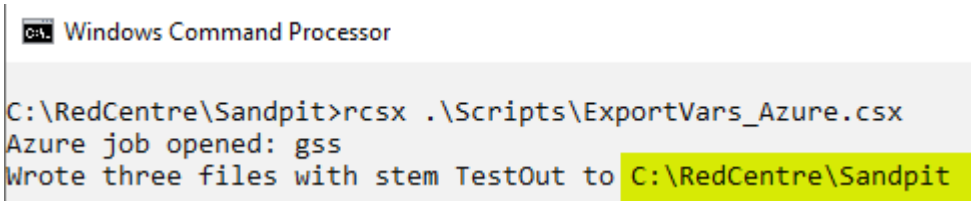
```
// Connect to Azure cloud storage and generate a table
var jobname = "gss";                        // USA General Social Survey
var eng = new CrossTabEngine();
await eng.LoginId("16499372","C6H12O6");    // guest licence
eng.OpenJob("rcspublic", jobname);          // Azure access happens here
Console.WriteLine($"Azure job opened: {jobname}");
var dprops = new XDisplayProperties();
dprops.Output.Format = XOutputFormat.SSV;  // space formatted
var sprops = new XSpecProperties();
var ret = eng.GenTab("Tab1", "Race(cwf%;*)", "Degree(cwf%;*)", "", "", sprops, dprops);
Console.WriteLine(ret);
```



ExportVars_Azure.csx

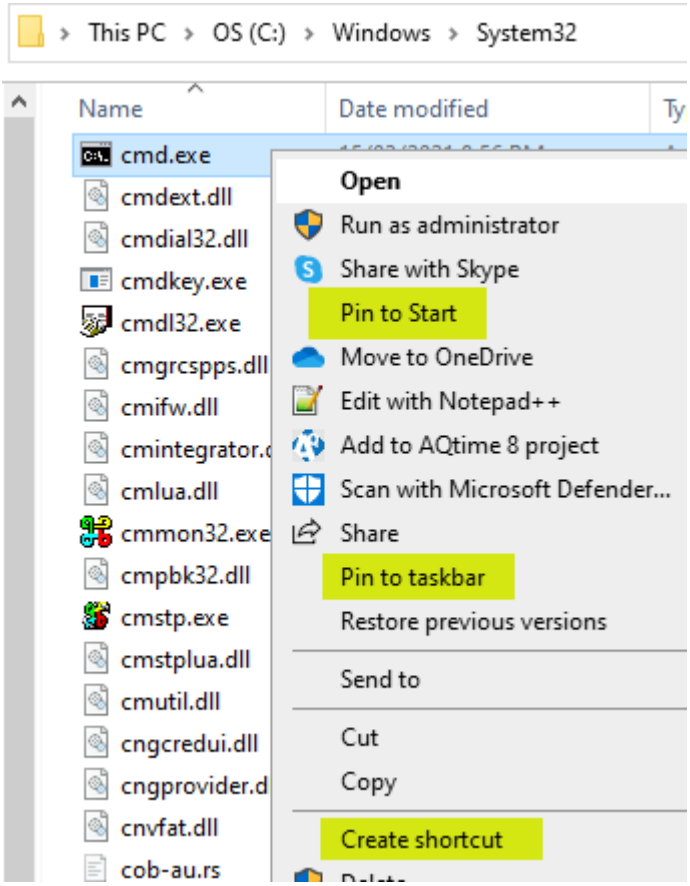
This example shows how to send outputs to a local directory.

```
var jobname = "gss";             // USA General Social Survey
var eng = new CrossTabEngine();
await eng.LoginId("16499372","C6H12O6");   // guest licence
eng.OpenJob("rcspublic", jobname);         // Azure access happens here
Console.WriteLine($"Azure job opened: {jobname}");

eng.SetHomeDir(@"C:\RedCentre\Sandpit");   // Azure data will arrive here

string exportvars = "race,Age8,sex";
var exp = new ExportEngine();
exp.AttachJob(eng);
var settings = new ExportSettings("TestOut")
{
    Format = VExportType.Tableau
};
var result = exp.ExportVars(exportvars, settings);
Console.WriteLine(result);
```



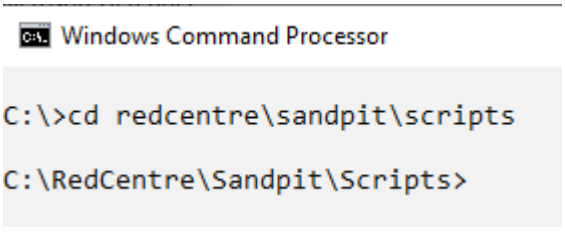
APPENDIX 1 COMMAND PROMPT

The Command Prompt is the old CLI (Command Line Interface, cmd.exe) from earlier Windows. Microsoft is trying to deprecate the command prompt in favour of PowerShell so it is no longer accessible from the Windows Start menu. The executable is C:\Windows\System32\cmd.exe.

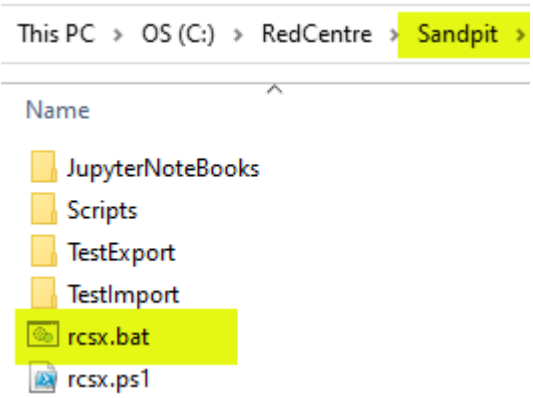


You can create a desktop or task bar shortcut in any of the usual ways.

One disadvantage is that there is no quick way to navigate to any directory and may take a long CD command, or a number of shorter ones, to get there.

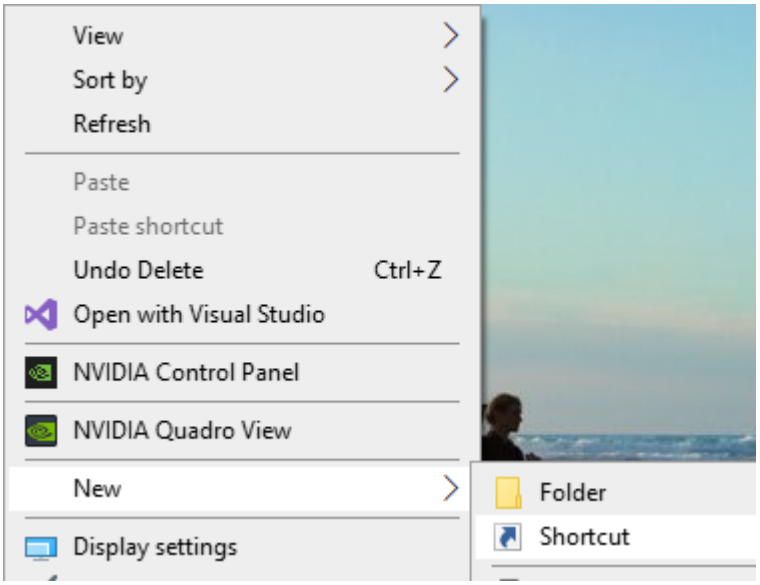


The Command Prompt looks in your current directory first, so as long as rcsx.bat is there …

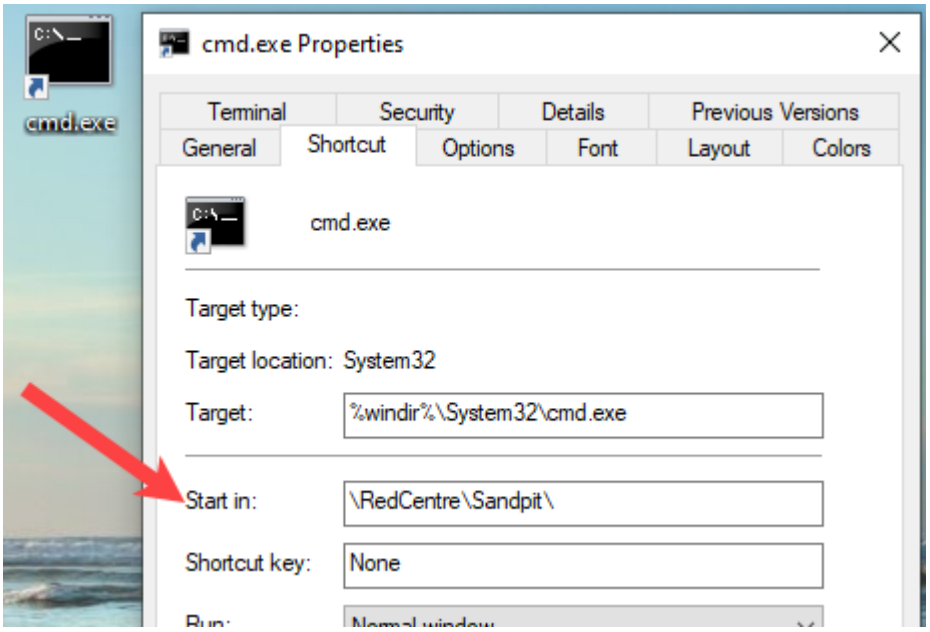


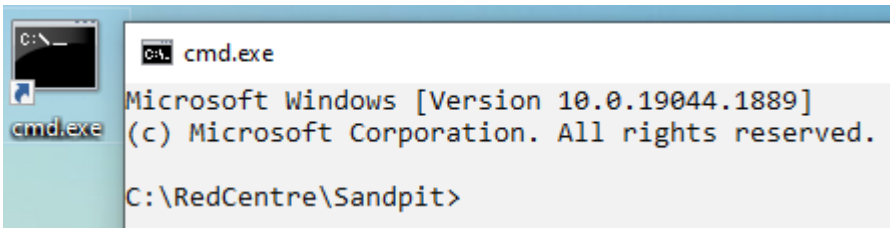
… the command >rcsx myscript.csx is all that is required.

You can avoid the navigation step by creating a desktop shortcut to cmd.exe.



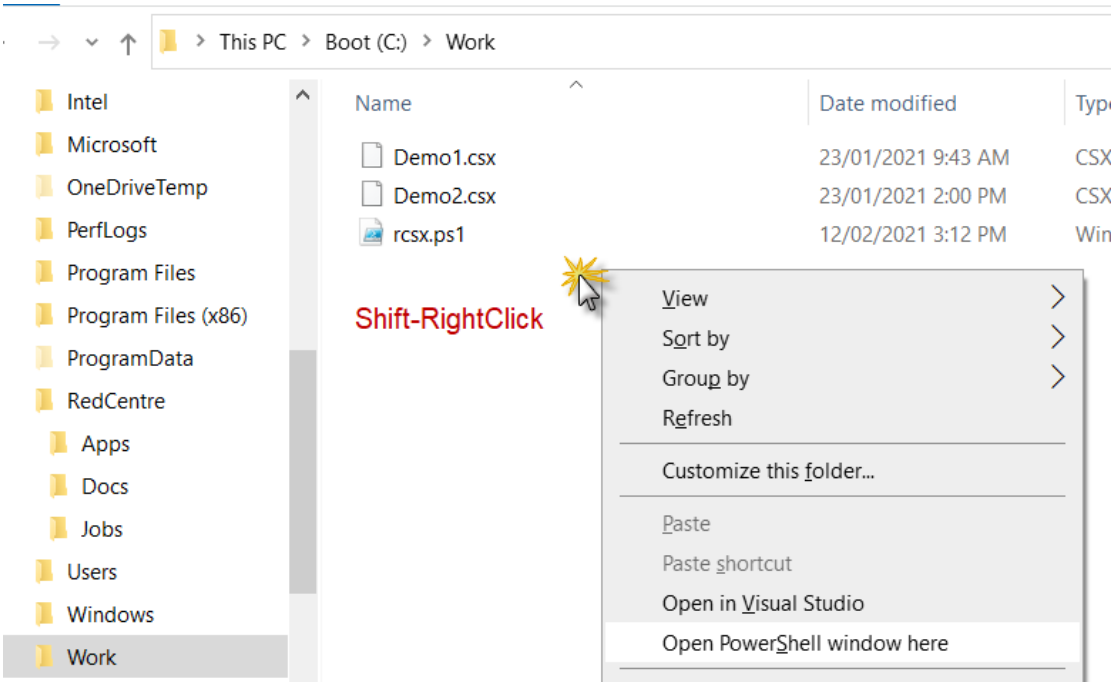
Then set the shortcut Start In property as your working directory.





APPENDIX 2 POWERSHELL

PowerShell is the official replacement for the old Command Line Interface, cmd.exe. One advantage is that it can be opened anywhere using Shift-Right Click in the whitespace of any directory.



One disadvantage is the fussy security. This is a shallow treatment of the topic so check with your IT manager on the preferred house policy.

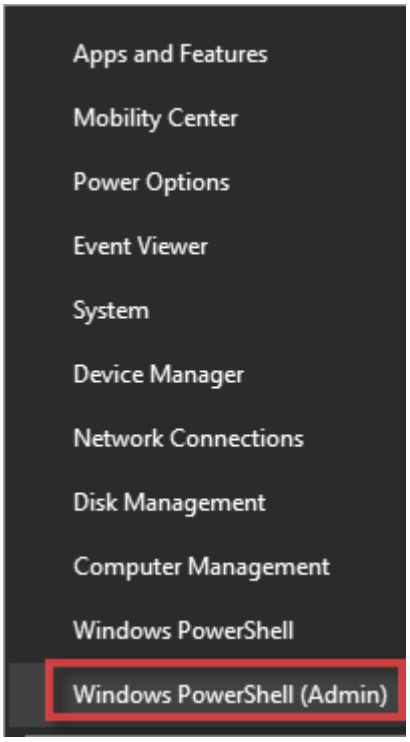
ExecutionPolicy

If you try >.\rcsx demo1.csx you are likely to get a fail saying rcsx.ps1 is unsigned. This is because the default policy for PowerShell is quite restrictive. You can see the policy with the command *Get-ExecutionPolicy -List*.

```
PS C:\WINDOWS\system32> get-executionpolicy -list

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine       AllSigned
```

There are several ways to allow script execution – your IT manager may prefer another way.

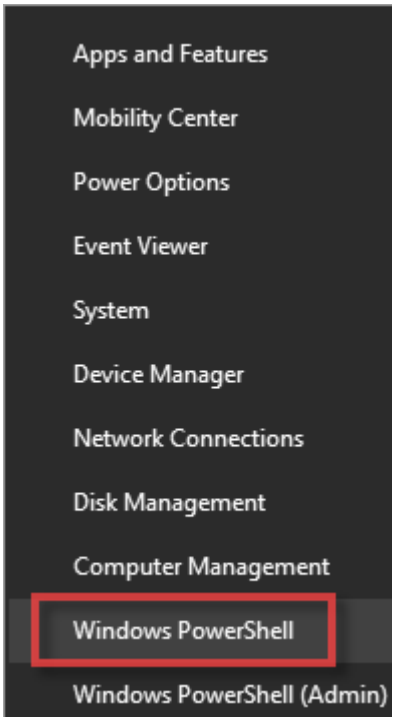
The simplest is to run PowerShell as Administrator from a right click on the Windows Start menu:

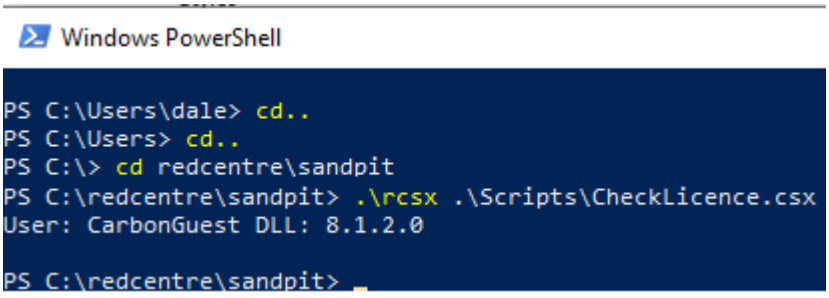


… and enter *Set-ExecutionPolicy Bypass* (note, no spaces around the hyphen).

```
PS C:\WINDOWS\system32> set-executionpolicy bypass

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic at
https:/go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"): y
PS C:\WINDOWS\system32>
```

You get a warning, type ‘y’ and press Enter. You can now use a standard PowerShell instance to run scripts.





Path

PowerShell does not look in your current directory for commands. You must enter the full path to rcsx.ps1 or use the short-cut path .\rcsx (in Windows, .\ means the current directory).

```
PS C:\Work> .\rcsx demo1.csx
Welcome CarbonGuest
```

PowerShell only looks on your current Path. You can see what directories are on your path with the command $Env:Path

```
PS C:\WINDOWS\system32> $env:path
C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\ProgramData\Oracle\Java
es (x86)\Embarcadero\Studio\21.0\bin;C:\Users\Public\Documents\Embarcadero\Studio\2
```

If you can copy rcsx.ps1 to one of these, or add say, c:\RedCentre\Apps to the path, then rcsx.ps1 will always be found by PowerShell and you will not need the .\ prefix.

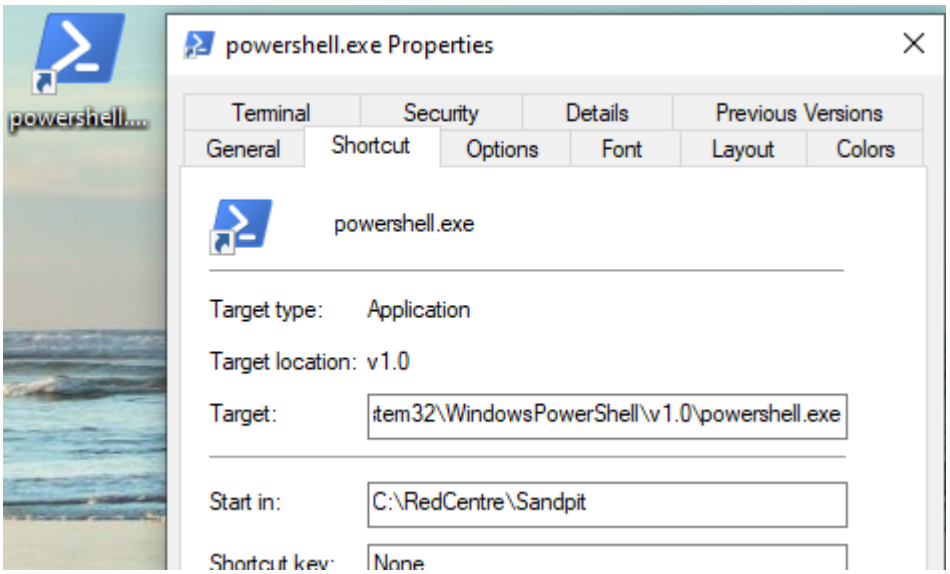
One handy directory always on your path is **C:\Users\<username>\AppData\Local\Microsoft\WindowsApps**. Copy rcsx.ps1 into there and it will always be found no matter what directory you are in.

```
PS C:\Work> rcsx demo1.csx
Welcome CarbonGuest
```

To be clear – you do not need to run PowerShell as Administrator for normal work.

Shortcut

As with cmd.exe, you can set up a desktop shortcut with a start folder, as



where the full Target path is

C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe

APPENDIX 3: ENGINES API

The main APIs for script writing are listed below.

CrossTabEngine

```
var eng = new CrossTabEngine();
```

Create an engine instance referenced as *eng*.

```
eng.VarTreefileNames()
```

Returns the names of .vtr files in the job directory.

```
eng.VarFrames(varname)
```

Returns the codeframe names in a variable.

```
eng.VarFrame(framename)
```

Returns the codes, expressions and nets in a codeframe. *framename* is the variable name for a simple variable or <variable name>.<level codeframe> for a hierarchic.

```
eng.CodeCount(framename)
```

Returns the number of codes in a codeframe.

```
eng.GenTab(name, top, side, flt, wght, sprops, dprops);
```

Runs a table using Ruby GenTab syntax. *sprops* and *dprops* are strongly typed objects (see GenTab examples) defining the runtime (sprops) and display (dprops) behaviour.

```
eng.SetProps(dprops)
```

Change display properties for the last run report.

```
eng.DisplayReport(OutputFormat.fmt)
```

Returns the last run table in a different format. *fmt* is one of the output formats: *None, TSV, CSV, SSV, HTML, OXT, XLSX*.

```
eng.OpenOutput(format,dest=null)
eng.OutputTable()
eng.OutputLine(s)
eng.CloseOutput()
eng.OutputMessage()
```

These are for managed table output going to a single file.

```
eng.Generate(varname)
```

Generate construction lines for a DefGen or DefGrid variable.

```
eng.Construct(varname)
```

Run a constructed variable to make the .CD data file.

```
eng.NewDefCon(name, description);
eng.NewDefCodeFrame(name, description);
eng.NewDefGen(name, filter, increment, description);
eng.NewDefGrid(name, desc, topname, top, sidename, side);
eng.NewDefNet(name, desc);
eng.NewDefWght(name, type, period, filter, rimdiff, pop, desc);
```

The DefCon family. See below for details.

```
eng.CopyVar(source, dest, desc, data);
```

source is existing variable; *dest* is new variable; *desc* is the description for the new variable; *data* is true/false meaning to copy/not copy the CD file as well.

```
string msg = eng.Message();
```

Return any error messages that may have accumulated.

ImportEngine

```
impeng.ImportDelim(filename, settings);
impeng.ImportSQLite(filename, settings);
impeng.ImportSQLiteQuery(filename, settings);
impeng.ImportTSAPI(filename, settings);
impeng.ImportSAV(filename, settings);
```

Import data from various formats. This should only be run in a new empty job. There is no support for wave imports yet.

The settings parameter is a typed object with member fields which control the import.

ExportEngine

```
expeng.ExportVars(varlist, settings);
```

Export data in tsv, csv, or Tableau format.

The settings parameter is a typed object with member fields which control the export.

APPENDIX 4: DEFCON FAMILY

Classes

DefCodeFrame

```
DefCodeFrame dc = eng.NewDefCodeFrame(name, description, "modify");
dc.AddCode(code,label);
dc.AddArith(exp,label);   // can be #c1+c2 or c1+c2
dc.AddNet(exp,label);     // must be _net(…)
dc.AddExpr(exp,label);
dc.AddItems(str)          \n sep set, labels
dc.RemoveItems(str )      \n sep set, tolerates labels
dc.Close()              // write the MET file
dc.Copy(frame, true/false) //copy codes, optionally arith and nets
```

DefCon

```
DefCon dc = eng.NewDefCon(name, description);
dc.AddItem(code, filter,label);           //code is a number
dc.AddInc(code, filter, increment, label);// code is a string and could be "*"
dc.AddExpr(arith/net, label);
dc.AddNextItem(filter, label);
dc.SetFlags(string);
dc.Close(build=false);
```

Note that the first parameter in *AddItem* is a number with no quotes and in *AddInc* it is a string. This is so you can use "*" for uncoded.

DefGen

```
DefGen dg = eng.NewDefGen(name, filter, increment, description);
dg.AddLevel(level, name, source, range, label);
dg.SetGenFlags(flags);
dg.Close(build=false);
```

Range can be blank meaning "*" and Label can be blank meaning "$<source>($)"

DefGrid

```
DefGrid dg = eng.NewDefGrid(name, desc, topname, top, sidename, side);
dg.Close(build=false);
```

DefWght

```
DefWght dw = eng.NewDefWght(name, type, period, filter, rimdiff, pop, desc);
dw.AddTarget(variable, targets);
dw.Close(build=false);
```

DefNet

```
DefNet dn = eng.NewDefNet(name,desc);
    DefNetLevel dl = dn.AddLevel(name);
        dl.AddItem(filter, label);
dn.Close(build=false);
```

DefDates

```
DefDates ddt = eng.NewDefDates(name, type, cycle, source, format, description);
ddt.Close(build=false);
```

Converting from Ruby to Carbon

Most of the Carbon objects are very similar to what a Ruby scripter would be familiar with so conversion will mostly be typographic details.

The wrapped variable methods in RedCentreLibrary.csx very closely match the RubyVariablesLibrary syntax. #load this at the top of your scripts to expose the wrappers.

```
#load @"\RedCentre\Apps\RedCentreLibrary.csx"
```

DefCodeFrame

In a Ruby script VB, you would have written:

```
DefCodeFrame("AidedBrandList", "")
  AddCode(1, "BrandX")
  AddCode(2, "BrandY")
  AddCode(3, "BrandZ")
CodeFrameClose()
Construct("AidedBrandList")
```

In Carbon CSX this becomes:

```
DefCodeFrame dcf = eng.NewDefCodeFrame("AidedBrandList", "");
  dcf.AddCode(1, "BrandX");
  dcf.AddCode(2, "BrandY");
  dcf.AddCode(3, "BrandZ");
dcf.Close(true);
```

Or if you #load RedCentreLibrary.csx, then the same as Ruby (except for trailing semi-colons and CodeframeClose(true)):

```
DefCodeFrame("AidedBrandList", "");
  AddCode(1, "BrandX");
  AddCode(2, "BrandY");
  AddCode(3, "BrandZ");
CodeFrameClose(true);
```

DefCon

In a Ruby script you would have written:

```
DefCon("S4_Group", "S4 Age Groups")
   AddItem(1, "S4(2;3)", "18yrs to 34yrs")
   AddItem(2, "S4(4;5)", "35yrs to 54yrs")
   AddItem(3, "S4(6;7)", "55yrs and above")
ConClose()
Construct("S4_Group")
```

In Carbon CSX this becomes:

```
DefCon dc1 = eng.NewDefCon("S4_Group", "S4 Age Groups");
  dc1.AddItem(1, "S4(2;3)", "18yrs to 34yrs");
  dc1.AddItem(2, "S4(4;5)", "35yrs to 54yrs");
  dc1.AddItem(3, "S4(6;7)", "55yrs and above");
dc1.Close(true);
```

Or if you #load RedCentreLibrary.csx it becomes:

```
DefCon("S4_Group", "S4 Age Groups");
  AddItem(1, "S4(2;3)", "18yrs to 34yrs");
  AddItem(2, "S4(4;5)", "35yrs to 54yrs");
  AddItem(3, "S4(6;7)", "55yrs and above");
ConClose(true);
```

DefGen

In a Ruby script you would have written:

```
DefGen("BF6_grid", "BF6_$a($b)", "Grid desc")
  AddLevel("a", "Top",  "BF6_List", "*")
  AddLevel("b", "Side", "BF6_1",    "*")
GenClose()
Construct("BF6_grid")
```

In Carbon CSX this becomes:

```
DefGen dg1 = eng.NewDefGen("BF6_grid", "BF6_$a($b)", "", "Grid desc");
  dg1.AddLevel("a", "Top",  "BF6_List");
  dg1.AddLevel("b", "Side", "BF6_1");
dg1.Close(true);
```

Or if you #load RedCentreLibrary.csx it becomes:

```
DefGen("BF6_grid", "BF6_$a($b)", "", "Grid desc");
  AddLevel("a", "Top",  "BF6_List", "*");
  AddLevel("b", "Side", "BF6_1",    "*");
GenClose(true);
```

Notice the extra third parameter. In Ruby there was *DefGen(name,filter,desc)* and *DefGenInc(name,filter,inc,desc).* In Carbon these have been combined so that *DefGen* ALWAYS takes four parameters including the increment one. If you forget to put the blank increment in a copied DefGen line you'll get an error like this

```
PS C:\Work> rcsx defcon.csx
defcon.csx(22,18): error CS7036: There is no argument given that cor
responds to the required formal parameter 'desc' of 'ManagedLocalJob
.NewDefGen(string, string, string, string)'
```

Notice also for AddLevel() you can leave out the "*" for Range as well as leaving out the Label. If blank, Range will be "*" and Label will be "$<source>($)"

DefGrid

In a Ruby script you would have written:

```
DefGrid("NPSS","copy of NPS", "Brand", "NPS_1;NPS_2;NPS_3", "Score","NPS_1")
GridClose()
```

```
Construct("NPSS")
```

In Carbon CSX this becomes:

```
DefGrid dg2 = eng.NewDefGrid("NPSS","copy of NPS", "Brand", "NPS_1;NPS_2;NPS_3",
"Score","NPS_1");
dg2.Close(true);
```

Or if you use RedCentreLibrary it becomes:

```
DefGrid("NPSS","copy of NPS", "Brand", "NPS_1;NPS_2;NPS_3", "Score","NPS_1");
GridClose(true);
```

DefWght

In a Ruby script you would have written:

```
DefWght("WghtAgeGenReg", "Cell", "Month", "Married(1)", 0, 0, "Demo weights")
  AddTarget("Age",    "10;30;30;20;10")
  AddTarget("Gender", "40;60")
  AddTarget("Region", "25;25;25;25")
WghtClose()
Construct("WghtAgeGenReg")
```

In Carbon CSX this becomes:

```
DefWght dw1 = eng.NewDefWght("WghtAgeGenReg", "Cell", "Month", "Married(1)", 0, 0,
"Demo weights");
  dw1.AddTarget("Age",    "10;30;30;20;10");
  dw1.AddTarget("Gender", "40;60");
  dw1.AddTarget("Region", "25;25;25;25");
dw1.Close(true);
```

Or if you use RedCentreLibrary it becomes:

```
DefWght("WghtAgeGenReg", "Cell", "Month", "Married(1)", 0, 0, "Demo weights");
  AddTarget("Age",    "10;30;30;20;10");
  AddTarget("Gender", "40;60");
  AddTarget("Region", "25;25;25;25");
WghtClose(true);
```

DefNet

In a Ruby script you would have written:

```
DefNet("Test6", "Example DefNet")
  AddNetLevel("Region")
      AddNetItem("region(1/2)",  "North")
      AddNetItem("region(3/4)",  "South")
  AddNetLevel("Work")
      AddNetItem("occupation(1/2)", "Professional")
      AddNetItem("occupation(2/3)", "Midlevel")
      AddNetItem("occupation(4/5)", "Other")
NetClose()
Construct("Test6")
```

In Carbon CSX this becomes:

```
DefNet dn = eng.NewDefNet("Test6", "Example DefNet");
  DefNetLevel dl = dn.AddLevel("Region");
      dl.AddItem("region(1/2)",  "North");
      dl.AddItem("region(3/4)",  "South");
  dl = dn.AddLevel("Work");
      dl.AddItem("occupation(1/2)", "Professional");
      dl.AddItem("occupation(2/3)", "Midlevel");
      dl.AddItem("occupation(4/5)", "Other");
dn.Close(true);
```

Or if you use RedCentreLibrary it becomes:

```
DefNet("Test6", "Example DefNet");
  AddNetLevel("Region");
      AddNetItem("region(1/2)", "North");
      AddNetItem("region(3/4)", "South");
  AddNetLevel("Work");
      AddNetItem("occupation(1/2)", "Professional");
      AddNetItem("occupation(2/3)", "Midlevel");
      AddNetItem("occupation(4/5)", "Other");
NetClose(true);
```

DefDates

In a Ruby script you would have written:

```
DefDates("Test7", "Date", "1y 1Jan2001", "Day",  "mmmyyyy", "Example DefDates")
DatesClose()
Construct("Test7")
```

In Carbon CSX this becomes:

```
DefDates ddt = eng.NewDefDates("Test7", "Date", "1y 1Jan2001", "Day",  "mmmyyyy",
"Example DefDates");
ddt.Close(true);
```

Or if you use RedCentreLibrary it becomes:

```
DefDates("Test7", "Date", "1y 1Jan2001", "Day",  "mmmyyyy", "Example DefDates");
DatesClose(true);
```

APPENDIX 5: PROPERTIES AND SETTINGS

The current set of spec and display properties with their default values:

Table Spec Properties

```
var sprops  = new SpecProperties()
{
   CaseFilter="",
   InitAsMissing=false,
   ExcludeNE=false,
   TopInsert="",
   SideInsert="",
   Level=""
};
```

For example,

```
SetJobDir(@"\RedCentre\Jobs\Demo");

sprops.CaseFilter="Day(1/500)";
sprops.InitAsMissing=false;
sprops.ExcludeNE=false;
sprops.TopInsert="cwf;cuf|tot%";
sprops.SideInsert="cwf;cuf|tot%;sgm%;nes";
sprops.Level="";

dprops.Output.Format = XOutputFormat.SSV;

var ret = eng.GenTab("test", "gender,married", "region", "", "", sprops, dprops);
Console.WriteLine(String.Join(Environment.NewLine, ret));
```

The case filter will ignore all days 1 to 500 (excluded from unfiltered base counts). TopInsert turns the top axis into `"gender(cwf;cuf;*tot%),married(cwf;cuf;*tot%)"`, and similarly the SideInsert turns the side axis into `region(cwf;cuf;*tot%;sgm%;nes)`.

Table Display Properties

```
var dprops  = new DisplayProperties();

Titles.Name.Visible=True
Titles.Name.Font.Name=
Titles.Name.Font.Size=10
Titles.Name.Font.Style=Regular
Titles.Name.Font.HAlign=Left
Titles.Name.Font.VAlign=Top
Titles.Name.Font.Color=255,0,0,0
Titles.Name.Font.Back=255,255,255,255
Titles.Name.Font.Wordwrap=False
Titles.Top.Visible=True
Titles.Top.Font.Name=
Titles.Top.Font.Size=10
Titles.Top.Font.Style=Regular
Titles.Top.Font.HAlign=Left
Titles.Top.Font.VAlign=Top
Titles.Top.Font.Color=255,0,0,0
Titles.Top.Font.Back=255,255,255,255
Titles.Top.Font.Wordwrap=False
Titles.Side.Visible=True
Titles.Side.Font.Name=
Titles.Side.Font.Size=10
Titles.Side.Font.Style=Regular
Titles.Side.Font.HAlign=Left
Titles.Side.Font.VAlign=Top
Titles.Side.Font.Color=255,0,0,0
Titles.Side.Font.Back=255,255,255,255
Titles.Side.Font.Wordwrap=False
Titles.Filter.Visible=False
Titles.Filter.Font.Name=
Titles.Filter.Font.Size=10
Titles.Filter.Font.Style=Regular
Titles.Filter.Font.HAlign=Left
Titles.Filter.Font.VAlign=Top
Titles.Filter.Font.Color=255,0,0,0
Titles.Filter.Font.Back=255,255,255,255
Titles.Filter.Font.Wordwrap=False
Titles.Weight.Visible=False
Titles.Weight.Font.Name=
Titles.Weight.Font.Size=10
Titles.Weight.Font.Style=Regular
Titles.Weight.Font.HAlign=Left
Titles.Weight.Font.VAlign=Top
Titles.Weight.Font.Color=255,0,0,0
Titles.Weight.Font.Back=255,255,255,255
Titles.Weight.Font.Wordwrap=False
Titles.Status.Visible=False
Titles.Status.Font.Name=
Titles.Status.Font.Size=10
Titles.Status.Font.Style=Regular
Titles.Status.Font.HAlign=Left
Titles.Status.Font.VAlign=Top
Titles.Status.Font.Color=255,0,0,0
```

```
Titles.Status.Font.Back=255,255,255,255
Titles.Status.Font.Wordwrap=False
Titles.Labelling.Script=False;
Titles.Labelling.Codes=False;
Titles.Labelling.Name=False;
Titles.Labelling.Desc=False;
Columns.Groups.Size=30
Columns.Groups.Visible=True
Columns.Groups.Font.Name=
Columns.Groups.Font.Size=10
Columns.Groups.Font.Style=Regular
Columns.Groups.Font.HAlign=Left
Columns.Groups.Font.VAlign=Top
Columns.Groups.Font.Color=255,0,0,0
Columns.Groups.Font.Back=255,255,255,255
Columns.Groups.Font.Wordwrap=False
Columns.Labels.Width=74
Columns.Labels.Height=67
Columns.Labels.Visible=True
Columns.Labels.Font.Name=
Columns.Labels.Font.Size=10
Columns.Labels.Font.Style=Regular
Columns.Labels.Font.HAlign=Left
Columns.Labels.Font.VAlign=Top
Columns.Labels.Font.Color=255,0,0,0
Columns.Labels.Font.Back=255,255,255,255
Columns.Labels.Font.Wordwrap=False
Columns.Letters.Font.Name=
Columns.Letters.Font.Size=10
Columns.Letters.Font.Style=Regular
Columns.Letters.Font.HAlign=Left
Columns.Letters.Font.VAlign=Top
Columns.Letters.Font.Color=255,0,0,0
Columns.Letters.Font.Back=255,255,255,255
Columns.Letters.Font.Wordwrap=False
Columns.Sort.Active=False
Columns.Sort.Increasing=False
Columns.Sort.Ungrouped=False
Columns.Sort.Band=First
Columns.Sort.Type=Value
Columns.Sort.Key=0
Columns.Hide.Active=False
Columns.Hide.Missing=False
Columns.Hide.Empty=False
Rows.Groups.Size=23
Rows.Groups.Visible=True
Rows.Groups.Font.Name=
Rows.Groups.Font.Size=10
Rows.Groups.Font.Style=Regular
Rows.Groups.Font.HAlign=Left
Rows.Groups.Font.VAlign=Top
Rows.Groups.Font.Color=255,0,0,0
Rows.Groups.Font.Back=255,255,255,255
Rows.Groups.Font.Wordwrap=False
Rows.Labels.Width=81
Rows.Labels.Height=20
Rows.Labels.Visible=True
Rows.Labels.Font.Name=
Rows.Labels.Font.Size=10
Rows.Labels.Font.Style=Regular
Rows.Labels.Font.HAlign=Left
```

```
Rows.Labels.Font.VAlign=Top
Rows.Labels.Font.Color=255,0,0,0
Rows.Labels.Font.Back=255,255,255,255
Rows.Labels.Font.Wordwrap=False
Rows.Letters.Font.Name=
Rows.Letters.Font.Size=10
Rows.Letters.Font.Style=Regular
Rows.Letters.Font.HAlign=Left
Rows.Letters.Font.VAlign=Top
Rows.Letters.Font.Color=255,0,0,0
Rows.Letters.Font.Back=255,255,255,255
Rows.Letters.Font.Wordwrap=False
Rows.Sort.Active=False
Rows.Sort.Increasing=False
Rows.Sort.Ungrouped=False
Rows.Sort.Band=First
Rows.Sort.Type=Value
Rows.Sort.Key=0
Rows.Hide.Active=False
Rows.Hide.Missing=False
Rows.Hide.Empty=False
Cells.Key.Font.Name=
Cells.Key.Font.Size=10
Cells.Key.Font.Style=Regular
Cells.Key.Font.HAlign=Left
Cells.Key.Font.VAlign=Top
Cells.Key.Font.Color=255,0,0,0
Cells.Key.Font.Back=255,255,255,255
Cells.Key.Font.Wordwrap=False
Cells.Bases.Font.Name=
Cells.Bases.Font.Size=10
Cells.Bases.Font.Style=Regular
Cells.Bases.Font.HAlign=Left
Cells.Bases.Font.VAlign=Top
Cells.Bases.Font.Color=255,0,0,0
Cells.Bases.Font.Back=255,255,255,255
Cells.Bases.Font.Wordwrap=False
Cells.Frequencies.Visible=True
Cells.Frequencies.Font.Name=
Cells.Frequencies.Font.Size=10
Cells.Frequencies.Font.Style=Regular
Cells.Frequencies.Font.HAlign=Left
Cells.Frequencies.Font.VAlign=Top
Cells.Frequencies.Font.Color=255,0,0,0
Cells.Frequencies.Font.Back=255,255,255,255
Cells.Frequencies.Font.Wordwrap=False
Cells.ColumnPercents.Visible=True
Cells.ColumnPercents.Font.Name=
Cells.ColumnPercents.Font.Size=10
Cells.ColumnPercents.Font.Style=Regular
Cells.ColumnPercents.Font.HAlign=Left
Cells.ColumnPercents.Font.VAlign=Top
Cells.ColumnPercents.Font.Color=255,0,0,0
Cells.ColumnPercents.Font.Back=255,255,255,255
Cells.ColumnPercents.Font.Wordwrap=False
Cells.RowPercents.Visible=True
Cells.RowPercents.Font.Name=
Cells.RowPercents.Font.Size=10
Cells.RowPercents.Font.Style=Regular
Cells.RowPercents.Font.HAlign=Left
Cells.RowPercents.Font.VAlign=Top
```

```
Cells.RowPercents.Font.Color=255,0,0,0
Cells.RowPercents.Font.Back=255,255,255,255
Cells.RowPercents.Font.Wordwrap=False
Cells.Stat1.Visible=False
Cells.Stat1.Font.Name=
Cells.Stat1.Font.Size=10
Cells.Stat1.Font.Style=Regular
Cells.Stat1.Font.HAlign=Left
Cells.Stat1.Font.VAlign=Top
Cells.Stat1.Font.Color=255,0,0,0
Cells.Stat1.Font.Back=255,255,255,255
Cells.Stat1.Font.Wordwrap=False
Cells.Stat2.Visible=False
Cells.Stat2.Font.Name=
Cells.Stat2.Font.Size=10
Cells.Stat2.Font.Style=Regular
Cells.Stat2.Font.HAlign=Left
Cells.Stat2.Font.VAlign=Top
Cells.Stat2.Font.Color=255,0,0,0
Cells.Stat2.Font.Back=255,255,255,255
Cells.Stat2.Font.Wordwrap=False
Cells.PercentSign.Visible=True
Cells.MissingAsZero=False
Cells.MissingAsBlank=False
Cells.ZeroAsBlank=False
Cells.PercentsAsProportions=False
Cells.ShowRedundant100=False
Significance.Visible=False
Significance.Type=SingleCell
Significance.PropStat=Z
Significance.MeanStat=T
Significance.StatsParam=
Significance.LetterSequence=
Significance.Letters=None
Significance.AppendLetters=False
Significance.Headcount=Weighted
Significance.MeanPoolVariance=False
Significance.PropPooledEst=False
Significance.ContinuityCorr=False
Significance.SkipBase30=True
Significance.SkipCell5=True
Significance.Letters64=False
Significance.OneTailed=False
Significance.SigLevel1.Threshold=95
Significance.SigLevel1.Font.Name=
Significance.SigLevel1.Font.Size=10
Significance.SigLevel1.Font.Style=Regular
Significance.SigLevel1.Font.HAlign=Left
Significance.SigLevel1.Font.VAlign=Top
Significance.SigLevel1.Font.Color=255,255,0,0
Significance.SigLevel1.Font.Back=255,255,255,255
Significance.SigLevel1.Font.Wordwrap=False
Significance.SigLevel2.Threshold=90
Significance.SigLevel2.Font.Name=
Significance.SigLevel2.Font.Size=10
Significance.SigLevel2.Font.Style=Regular
Significance.SigLevel2.Font.HAlign=Left
Significance.SigLevel2.Font.VAlign=Top
Significance.SigLevel2.Font.Color=255,0,0,255
Significance.SigLevel2.Font.Back=255,255,255,255
Significance.SigLevel2.Font.Wordwrap=False
```

```
Significance.SigLevel3.Threshold=80
Significance.SigLevel3.Font.Name=
Significance.SigLevel3.Font.Size=10
Significance.SigLevel3.Font.Style=Regular
Significance.SigLevel3.Font.HAlign=Left
Significance.SigLevel3.Font.VAlign=Top
Significance.SigLevel3.Font.Color=255,0,255,0
Significance.SigLevel3.Font.Back=255,255,255,255
Significance.SigLevel3.Font.Wordwrap=False
Decimals.Frequencies=0
Decimals.Percents=2
Decimals.Statistics=1
Decimals.Expressions=2
Output.Format=None
Corner.Priority=Side
```

Output.Format can be *None*, *TSV*, *CSV*, *SSV*, *HTML*, *OXT* or *XLSX*.

All table output formats produce a text response except XLSX, which writes the current reports as <name>.xlsx in the job/Docs directory.

Export Settings

```
var settings = new ExportSettings(exportFilename)
{
   BinaryMulti = false,
   CodeLabels = true,
   FirstLineIsVars = true,
   QuoteNumeric = false,
   QuoteText = true,
   Format = EngineExportType.TSV
};
```

EngineExportType can be *TSV*, *CSV* or *Tableau*.

Import Settings

There are different settings objects for each import type. For SAV files

```
var settings = new ImportSavSetting
{
   Overwrite=true,
   TryBlend=true,
   ZeroBased=false
};
```

Examples of the other import types are

Delimited

```
string src = @"D:\1jobs\rcs\testimport\source\2001\minidemo.csv";
ImportDelimSettings sett = new ImportDelimSettings();
sett.Type = ImportDelimType.CSV;        // or TSV
string res = imp.ImportDelim(src, sett);
```

TSAPI

```
const string BaseAddress = "https://tsapi-demo.azurewebsites.net/";
var surveys = await imp.ListTSAPISurveys(BaseAddress);
```

```
    string s = await imp.ImportTSAPI(BaseAddress, surveys.First().Id);
```

SQLite

```
    ImportSettings sett = new ImportSettings();
    sett.ZeroBased = false;
    string connstring = @"Data Source=D:\1jobs\rdb\twit\recent100.db3";
    string s = imp.ImportSQLite(connstring, "Tweets", sett);
```

SQLiteQuery

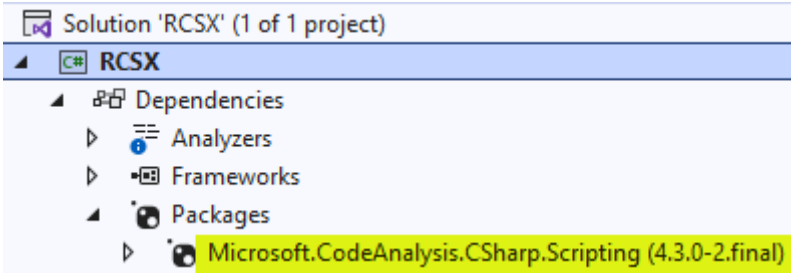
```
    ImportSettings sett = new ImportSettings();
    sett.ZeroBased = false;
    string connstring = @"Data Source=D:\1jobs\rdb\twit\recent100.db3";
    string query = "Select CreatedAt as TimeDone, CreatedBy as DoneBy, Cast(CASE
  WHEN IsRetweet='0' THEN 'False' ELSE 'True' END as TEXT) as Retweet from
  Tweets Limit 100";
    string s = imp.ImportSQLiteQuery(connstring, query, sett);
```

APPENDIX 6 – RCSX.EXE

As noted above, the official CSX compiler csi.exe does not work with Net6, and since

i) Net6 is about 30% faster than Net Framework (versions 4.x and earlier) and
ii) Microsoft has no timeline for making csi.exe Net6 compatible

the only recourse was to write our own wrapper for the Net6 C# Scripting package, which we call *rcsx.exe* (for Run CSX).



This raised both problems and opportunities.

The main problem is that the CSharpScript object executes a string, not a file, and so does not handle relative paths (at least, not well enough for our requirements).

The work-around is to expand #r and #load relative paths to absolute while reading the file, and then pass the expanded text as CSharpScript.EvaluateAsync(expanded_csx_text, options).

A second problem is that error reporting does not know the name of the calling script:

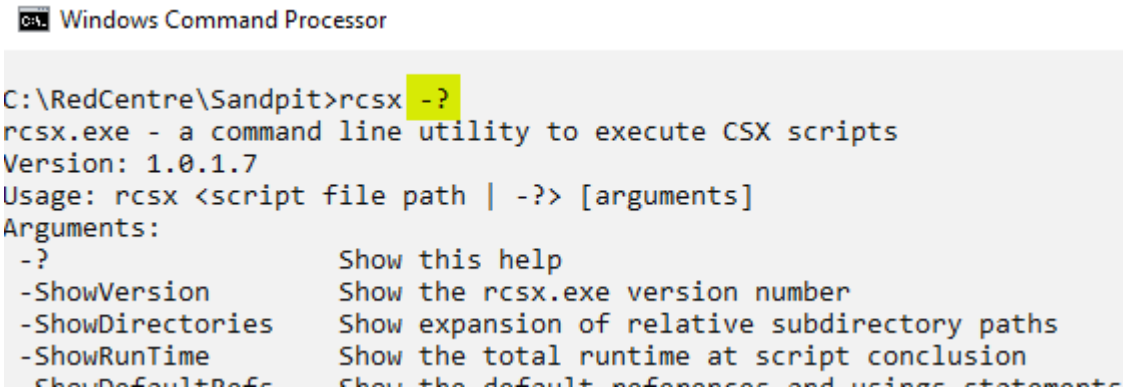
```
C:\RedCentre\Sandpit>rcsx .\Scripts\DeliberateError.csx

C:\RedCentre\Sandpit>C:\RedCentre\Apps\RCSX\rcsx.exe .\Scripts\DeliberateError.csx
Error in script file: C:\RedCentre\Sandpit\Scripts\DeliberateError.csx
C:\RedCentre\Sandpit\Scripts(6,11): error CS0103: The name 'engXXX' does not exist in the current context
```

The yellow highlight gives the correct coordinates for line and column (as 6, 11), but does not say of which file. The work-around is the grey highlight line, put there by rcsx.exe itself.

The opportunities are CLI arguments and default references.

Arguments

The first argument is always either the script name or -? for help.



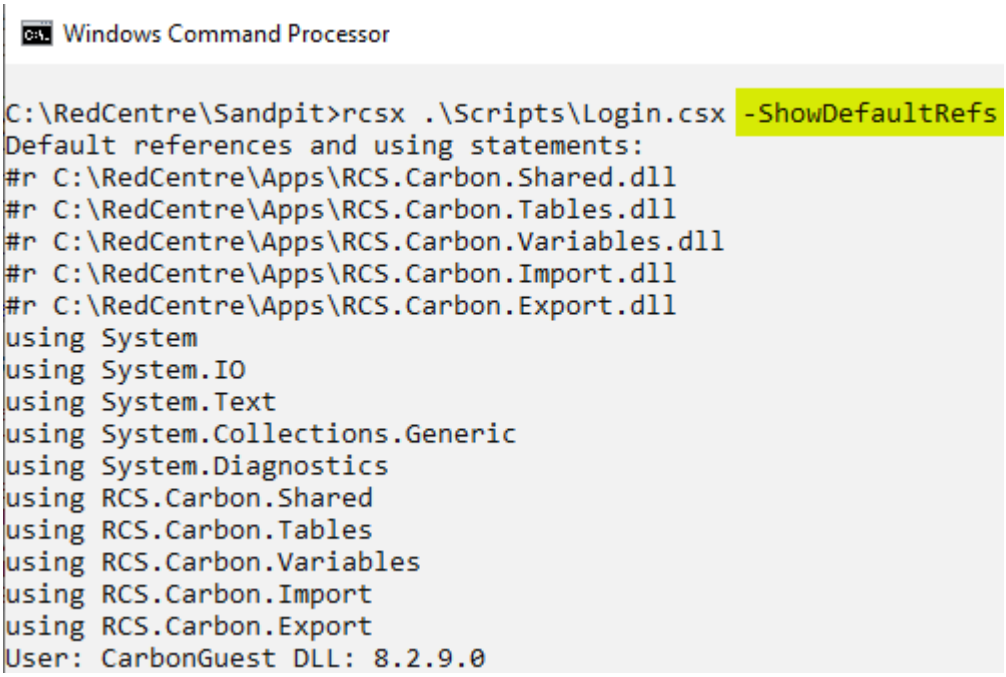
The other arguments are optional flags.

-ShowVersion	show the rcsx.exe version number
-ShowDirectories	show #r or #load relative path expansions (this is a diagnostic)
-ShowRuntime	show the script execution time
-ShowDefaultRefs	show the default references and usings statements
-NoDefaultRefs	the script must have explicit #r references and usings statements
-RCSReferencePath@	set the references drive and path to \\RedCentre\\Apps,");

The first three -Show flags can be applied to any script.

The idea is to have clean output as the default behaviour, and then to progressively clutter it up as may be required.

-NoDefaultRefs means that you will need to cite the references and usings statements in your script. If you browse the top level scripts, you will see none has any usings statements. This is because the references and usings are by default passed to the CSharpScript object. To see the defaults, run as



Use -NoDefaultRefs if you want to write C# compatible scripts for execution using Visual Studio or VS Code (so you can copy/paste between C# and CSX).

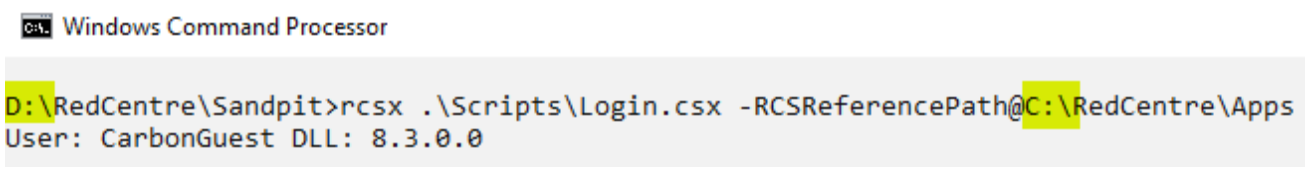
The example files are Startup_NoDefaultRefs.csx and ImportDemoDems_NoDefaultRefs.csx. The startup script has

```
// Expects -NoDefaultRefs
using System;
using RCS.Carbon.Tables;

var eng = new CrossTabEngine();
...
...
```

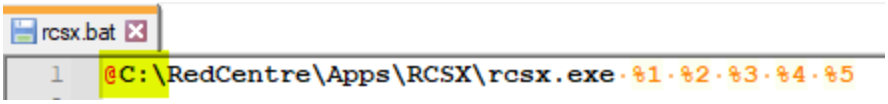
And the import script needs

```
#r "..\..\Apps\RCS.Carbon.Shared.dll"
#r "..\..\Apps\RCS.Carbon.Tables.dll"
#r "..\..\Apps\RCS.Carbon.Import.dll"
#load "..\startup_NoDefaultRefs.csx"

using System;              // console
using RCS.Carbon.Import;   // Import engine

var jobdir = @"\RedCentre\Sandpit\TestImport";
...
...
```

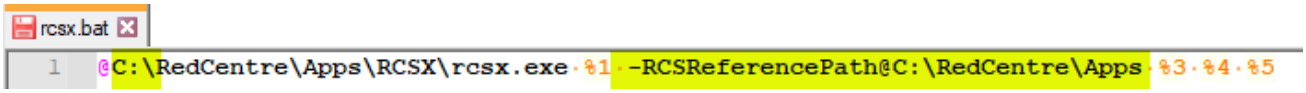
Use -RCSReferencePath if you want to reference the DLLs on a different drive to the current working directory. If your job is on drive D:\ and \RedCentre\Apps is on drive C:\, then



For this to work, D:\RedCentre\Sandpit\rcsx.bat must also look for rcsx.exe on drive C:\



And if editing rcsx.bat in this scenario, then the argument can be pre-supplied in the second position as



[End of document]