**An APL-Excel Interface:
User Guide**

Author: David Crossley

Date: September 22, 2011

© David Crossley 2010, 2011

davidwfc@gmail.com

Table of Contents

[Overview 2](#_Toc304464369)

[About the Toolkit 2](#_Toc304464370)

[Concepts 2](#_Toc304464371)

[Toolkit Functions and Arguments 3](#_Toc304464372)

[Excel Terms 3](#_Toc304464373)

[Comparing VBA to Apl 4](#_Toc304464374)

[Installation 4](#_Toc304464375)

[Basic Concepts 4](#_Toc304464376)

[Getting Going 4](#_Toc304464377)

[Some Pitfalls 5](#_Toc304464378)

[Cleaning Up 6](#_Toc304464379)

[Sudoku – An Illustrative Exercise 7](#_Toc304464380)

[Drawing the Grid 7](#_Toc304464381)

[Defining a Puzzle 10](#_Toc304464382)

[Reading Values, Formulas, Text and Dates 13](#_Toc304464383)

[Reading a Dynamic Range 14](#_Toc304464384)

[Writing Values, Formulas and Dates 15](#_Toc304464385)

[xread and xwrite 17](#_Toc304464386)

[Plotting 17](#_Toc304464387)

[Appendix 20](#_Toc304464388)

# Overview

APL and Excel are similar applications – they are both interpreters with visible sessions in which you can enter data and run programs, and both are very effective, in different ways, in the way they handle arrays. They are complementary, however,in that Excel’s strength is the display of data while Apl’s strength is in the manipulation and computation of data. Given this, and given the popularity of Excel, it makes very good sense to link the two environments together, thereby significantly extending the problem domain of both applications. There are in fact three ways to link them:

 Copy and paste between Apl and Excel sessions

 Call an Apl program from an Excel Program

 Execute Excel commands in the Apl environment.

The third way is the subject of this toolkit.

This tookit covers only a small number of the operations that one may wish to perform when working with Excel and will never cover all Excel methods – there are far too many. Some functions are quite brief, but offer the advantage of a short name instead of a long and moderately complex Excel command; some functions are quite long and provide much more value added. The relatively small scope of the toolkit means that you may need to write Excel commands explicitly. Understanding at least some parts of the Excel Object Model is therefore very helpful and a few examples of programming Excel directly are included in this guide as well as in the Reference Manual.

Just because you can program Excel in the Apl environment doesn’t always mean that you should. A program in the Excel environment may be more appropriate than an external program and VBA (Visual Basic for Applications) programs can interact with Excel more quickly than Apl programs, though in many cases you won’t notice the difference. However, if you know Apl better than VBA, this toolkit will help you to get started in terms of linking the two environments.

# About the Toolkit

The toolkit was developed and tested originally in Dyalog versions 12.1 and 13.0 using Excel 12.0 (Microsoft Office 2007). If you use an earlier version of Excel, basic read and write functionality hasn’t changed but some aspects of plotting have changed and there may be other incompatibilities. If you use Dyalog 11 or 12, you **must** set []WX🡨1 globally because of a bug in those versions. The toolkit does not work in Dyalog 10.1 (which does not support indexing of collections).

## Concepts

The overriding goal of the toolkit is to provide a basic and simple interaction with Excel. With that in mind, these concepts follow:

* So that you don’t have to specify workbook and worksheet every time you specify an Excel range, all functions except *xread* and *xwrite* apply to the active workbook or active worksheet (see below) where appropriate.
* For the sake of programmatic usage, all functions allow Excel ranges as row-column co-ordinate pairs as well as the ‘A1:B5’ type of notation. Functions *rc* and *rd* convert between these forms. The examples in the documentation may show either or both types of usage.
* The functions must be simple enough that users can understand the Excel components and hence add their own functions using the existing functions as models. This concept is the main justification for not overburdening the functions with argument-checking and error-trapping. Most functions are, therefore, short and quick – many are dynamic functions – but lack robustness.

While using the toolkit, the Apl environment is always in sync with the Excel session. This means, for example, that if you open a workbook manually in Excel, the new workbook is immediately known to be the active workbook in the Apl environment. Similarly, if you activate any sheet in any workbook in the Apl environment, that sheet is the one that now has focus in the Excel session. Without exception, whatever change you make in one environment is immediately knowable in the other.

## Toolkit Functions and Arguments

The toolkit functions are grouped by category in the Appendix. You can obtain one line of ‘help’ information for each function by executing *help* (no arguments); you can obtain help information for any single function except *help*, *lc*, and *clean* by supplying ‘?’ as the function’s right argument. For further information, please see the Reference Manual. The price of allowing ‘?’ is that you have to remember to type an argument (such as ‘’) for several functions even though the argument contributes nothing to its execution.

Rules for workbook name, sheet name and cell range as elements for function arguments are described below. All three are case-independent but all other character string arguments must be specified in lower case.

**Workbook name**:

* When you open a workbook, or save a workbook under another name, you must specify the name and path.
* When you refer to an open workbook, always omit the path (this is not important for functions *xread* and *xwrite*). You can also specify an open worbook by number, which is its index in the Excel Workbooks Collection, and you can specify '' (an empty string) to specify the active workbook.
* Where you specify a name, the file extension can usually be omitted but if there are two files of the same name with different extensions then it is sensible to specify the extension; if you don’t, Excel will apply its own rules in deciding which file to use.

**Sheet name**: You can specify the name or its number, which is an index to the workbook’s Sheets Collection. You can also specify '' (an empty string) to specify the active worksheet. Visually, sheet *n* is the *n*th sheet from the left (assuming all sheets are visible). Beware that a sheet named ‘Sheet3’, for example, is not necessarily the 3rd sheet.

**Cell range**: Two types of cell addresses are allowed: the standard A1:B2 style and row-column co-ordinates, which are often preferable for programming. Blanks are removed from address strings (Excel doesn’t allow them). Co-ordinates for a single cell must be parenthesized if they are part of a list. Here are some examples:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| String: | ‘b6’ | ‘G 8’ | ‘b6: g 8’ | ‘H4:A2’ |
| Co-ordinates: | 6 2 | 8 7 | (6 2)(8 7) | (4 8)(2 1) |

The toolkit does not currently support named ranges. If you specify a named range or table, just the top left corner will be used.

## Using Excel Objects

Some functions return an Excel object. This is very helpful because it means that certain methods and properties need not be implemented as Apl functions since you can invoke them directly from the object in Apl. As with all objects in Apl, the Excel objects have MethodList and PropList properties. In the list of workbook methods, for example, you will see Save and Close. Thus you can use the expressions *wbk.Save* and *wbk.Close ‘’* (note the required argument) where *wbk* is an object returned by *open* or *setbook*. However, an Apl function is helpful if you want to close the workbook in a way that deals with unsaved changes without causing an Excel prompt.

Your experience of using Excel will be greatly enhanced if you become familiar with some of the Excel objects and their properties and methods. Some examples are given in the section “Excel Properties and Methods” in the Referencemanual.

## Excel Terminology

Two important Excel terms are Active Worksheet and Active Workbook. The former is a reference (and ref, in Apl jargon, is very appropriate) to the sheet that has focus in the Excel session. Let’s say that the active sheet is sheet 1; if you then select sheet 3 in Excel, sheet 3 becomes the active sheet and the properties of xl.ActiveSheet – the active sheet object – immediately change.

For the many functions that take only a cell range as an argument, such as *read*, *write* and *format*, the source or target of the operation is always the active sheet. You can designate another sheet as ‘active’ either by selecting another sheet in Excel or via the function *setsheet*.

The Active Workbook is the workbook that owns the active sheet. By opening or activating another workbook, you change both the active workbook as well as the active sheet. You can select another open workbook via *setbook* and you can open a new book via *open*. and you can execute *active ''* to see what are the active book and sheet. Opening and closing workbooks and deleting or copying sheets generally do change what’s active. You can see the names of the active book and sheet by executing *active ''*.

# Comparing VBA to Apl

In principle, any command you can execute via VBA you can also execute via APL. In practice, however, there are difficulties in translating from VBA to Apl. For example:

* Apl has a different syntax (for example, parentheses are used differently and named parameters are specified differently).
* Apl requires a path prefix for variables (except dynamic function arguments) used in constructing Excel objects whereas VBA doesn’t require a prefix. For example, xl.(Range r).Value2 fails unless you prefix r by its path; alternatively, you can use a dynamic function with r as the argument. In particular, use of variables inside Apl *:With* blocks can be tricky.
* Some commands can be abbreviated in VBA but abbreviation fails in Apl (eg, in VBA you can omit “.Item” when referring to a collection but it’s mandatory in Apl unless you use indexing).
* Some VBA commands need a lot of experimentation (creating a chart is one example) before a construct can be found that works in Apl.
* Some commands, such as the format painter, I have not been able to implement in Apl.

The VBA editor has some benefits which Apl lacks:

* It shows the list of an object’s properties and methods even if the object doesn’t exist.
* The list appears regardless of the number of periods in the path.
* When you specify a method, a tool tip appears which shows you the method’s parameters.

Overall, VBA provides a simpler programming environment than Apl for manipulating Excel objects. The difficulties of translation to Apl, in addition to the fact that not every Apl programmer knows Excel VBA intimately, justifies the creation of the Apl-Excel interface functions described here. Further, some functions in this toolkit provide significant value-added.

# Installation

The directory that contains this document contains three files with a .dyalog extension:

load\_ws.dyalog

xlc.1.dyalog for Dyalog Classic versions

xlu.1.dyalog for Dyalog Unicode versions

In Apl:

To install the toolkit in an application workspace that you have already created,

1. )load the workspace
2. )cs
3. ⎕← ⎕SE.SALT.Load'd:\apl\AEI\xlu' ⍝unicode version

#.XL

1. )save

Step (2) isn’t essential but if the namespace is owned by root it shortens the path for the occasions when you need to type it explicitly. If you want the classic version, specify ‘xlc’ instead of ‘xlu’ in (3).

To use the toolkit, you should do this now and each time you start Apl:

 ⎕path,←' #.XL'

You should specify the output of (3) instead of ‘#.XL’ if you installed the toolkit somewhere other than #.XL.

# Basic Concepts

## Getting Going

The first action that must be performed before Apl can ineract with Excel is to create the Excel object model in the Apl environment using the Excel OLEServer. Taking a line from the function *startexcel*, this is how to achieve that:

 'xl' ⎕WC 'oleclient' 'excel.application'

‘xl’ is an arbitrary name that ⎕WC creates; you could instead use ‘e’ or ‘EXCEL’ or anything else. *xl* is an obect that exposes the entire Excel Object Model – a huge collection of objects, methods and properties. This object model is exactly the same as what is available when you write programs in VBA (Visual Basic for Applications). Thus anything you can achieve in VBA you can also achieve in Apl – beware, however, that Apl has different syntactical constructions, so you cannot, in general, just copy a VBA command into Apl and run it.

If Excel is not running, the above command starts Excel. If Excel is running, the command just attaches to an Excel session. (Unfortunately, if there are two or more instances of Excel running, it’s not possible to specify which one it will connect to.) By “attaching to” an Excel session, we mean that *xl* becomes a representation of the visual Excel session that is always in sync with the visual session. This means, for example, that if you open a workbook manually in Excel, the new workbook is immediately known (via *xl*) to be the active workbook in the Apl environment. Similarly, if you activate a worksheet via *xl*, that sheet is the one that now has focus in the Excel session. Without exception, whatever change you make in one environment is immediately knowable in the other.

There will be some occasions where you want to create a new Excel session rather than attach to an existing session. In that case, the initial command is:

 'xl' ⎕WC 'oleclient' 'excel.application' 'new'

Enven if *xl* were attached to an Excel session before, this command re-creates *xl* so that it isnow linked to the new session.

Like all other objects, xl has MethodList and PropList properties. If you are familiar with Excel, displaying these via #.XL.xl.MethodList or #.XL.xl.PropList will display many familiar names. In particular, you can set the Visible property in order to show or hide the Excel window. So, in order to stop yourself from inadvertently using an Excel session that your program is using, you can hide it by:

 #.XL.xl.Visible ← 0

In the context of this toolkit, you should use *startexcel*, which invokes the above commands as appropriate. The syntax is:

 R 🡨 {α} startexcel ω

where

ω is an empty vector or ‘new’ which determines whether to use an existing Excel session, if possible, or else create a new one

α is 0 or 1 (=show, the default) which controls the Visible property.

If *startexcel* is successful, R is zero but an error is trapped (error number 500 is used for all traps in the toolkit). If you forget the argument values, type this to obtain a reminder:

startexcel '?'

start Excel; ω: '' to use existing Excel session, if possible, or 'new' to start new Excel sesssion; α: 1=visible (default), 0=hidden

Almost every function in this toolkit allows ‘?’ as a right argument.

*startexcel* performs one other task: it defines ‘wb’ as an object known as the Workbooks Collection. In essence, this is a list of all the open workbooks, though which you can access any workbook’s properties and methods. *wb* is defined by:

wb ← xl.Workbooks

This is a trivial expression but it’s used frequently so it’s helpful to have a short name. With wb, you can type #.XL.wb.C, for example, to see the pop-up window showing all the properties and methods beginning with C, which wouldn’t happen if you typed #.XL.xl.Workbooks.C.

A useful property is wb.Count, which tells you how many books are open. The function *books* (executed as *books ''*) uses wb.Count and returns a matrix of names of all the open workbooks.

## Some Pitfalls

It is very easy to trip up with this toolkit and, when you do, it’s seldom obvious what the problem is. Here are some examples of errors which might help you resolve a problem:

1. If Excel is busy (a macro is running) or is awaiting user input, either in a dialog box or in a cell, the instruction

startexcel ''

wont’t return. You can wait for the macro to finish, find and close the dialog box, or complete cell input. Another option is to interrupt *startexcel* and then start a new session:

startexcel 'new'

1. These two examples illustrate what can happen if you forget to execute *startexcel*:

books''

Value Error: wb probably not defined - execute ∇startexcel

read'a4:b8'

VALUE ERROR

rc[6] (xl.Range addr).(Row Column)

1. If you have terminated Excel since executing *startexcel*, or if you )load a workspace containing Excel objects, *xl* and *wb* are now dead objects and so:

active ''

SYNTAX ERROR

active[2] xl.ActiveWorkbook.(Name FullName),⊂xl.ActiveSheet.Name

 read 'a4:b8'

DOMAIN ERROR

rc[6] (xl.Range addr).(Row Column)

In addition, you will see a dialog box like this:

 

1. You type an incorrect name when referring to an open workbook:

setbook 'mistake'

DOMAIN ERROR

setbook[3] aw←wb.Item ⍵ ⍝Item is more tolerant than []

1. You type a bad address:

read '5a:p12'

DOMAIN ERROR

rc[6] (xl.Range addr).(Row Column)

This also causes a dialog box like the one above.

Even though errors can be trapped, that does not prevent the display of dialog boxes. Note that the functions *xread* and *xwrite* anticipate errors like these.

## Cleaning Up

All Excel objects in Apl, such as *xl*, *wb* and any you create yourself, have meaning only when they are connected to an Excel session. If you terminate Excel from Excel itself or the Windows Task Manager, for example, the objects remain in the Apl workspace but they are no longer usable; effectively they are dead objects. You can also kill your objects as follows: save a workspace with objects that are still attached to an Excel session, then load it. In the just-loaded workspace, Apl tells you that you have some dead objects with a dialog box like this:



If this dialog box isdisplayed, click Close then erase the Excel objects by executing:

 endexcel ''

If you execute *endexcel* before you *)save* then you will avoid the dialog box.

*endexcel* erases Excel objects (both living and dead) from #.XL as well as from the namespace you are using when you invoke *endexcel*. However, any Excel objects in other namespaces will survive the purge. Despite its name, *endexcel* does **not** terminate the Excel session itself. If you want to terminate Excel too, execute:

 quitexcel ''

*quitexcel* terminates Excel first then invokes *endexcel*.

Warnings:

* If you invoke *endexcel* then *quitexcel*, *quitexcel* cannot now terminate Excel.
* If there is an unsaved workbook when you execute *quitexcel*, Excel will display a dialog box (“Do you want to save …”). Apl does not know about this and you may not see it, so if you subsequently execute *startexcel* (without the ‘new’ option) while the dialog box is open, *startexcel* will hang while it waits for Excel to respond.

# Sudoku – An Illustrative Exercise

Sudoku puzzles are common enough that a description of Sudoku will be omitted. The aim here is not to solve a Sudoku puzzle but to create a workbook of puzzles and to write a small Apl program to help you get started with a manual solution.

A few toolkit functions will be introduced so that you can develop a sense of how the toolkit can help you. The text does not cover all aspects of these functions so you will need to refer to the Reference Manual for complete information on these and other functions.

## Drawing the Grid

Let’s assume that Excel is not running. We therefore start it by:

 startexcel ''

0

Now let’s create a new workbook for this exercise:

 su←open ''

An empty vector as the right argument means create a new book; alternatively we could specify a file name including the path. We don’t have to assign the result of *open*, but it is often helpful. *su* is a workbook object; you can see its properties via su.PropList and you can obtain the workbook’s name, for example, by:

 su.Name

Book1

By default, the first sheet is the active sheet (the one that has focus in the Excel window). You can verify what’s active by:

 active ''

Book1 Book1 Sheet1

The three components of the result are workbook short name, workbook long name (includes path and suffix if it has been saved) and active sheet.

Now let’s draw the grid lines – thick borders around the 3-by-3 groupings and thin borders elsewhere. Our grid, however will contain 18 rows, not 9. We want to split each cell vertically into two: the bottom one will hold the number assigned to the cell, or else will be empty; the top one will show the digits allowed in the cell if the bottom one is empty. Here is an illustration of the goal (as close as I one get with Word):

|  |  |  |
| --- | --- | --- |
|   | 249 | 2349 |
| **7** |   |   |
| 34689 | 2469 | 234689 |
|   |   |   |
| 36 |   |   |
|   | 5 | **1** |

The top-left and bottom-right cells were in the initial definition of the puzzle so the font is bold and a background colour has been set. The cells above these are empty. The cell in the middleof the bottom row was defined manually, so neither bold nor background set. In all the other cells, the upper cells contain the allowed digits for the lower cells and are left-justified in a smaller font. Also note that there are no boundaries between the upper and lower cells – this is for visual clarity.

We will locate the grid with the top left corner at B2 so that the top and left borders are clear of the edges of the sheet. The range of the grid will therefore be B2:J19. We start by drawing a double-line border around the entire grid:

 9 frame 'b2:j19'

In Excel jargon, *frame* draws outside borders but no inside borders. Left argument 9 specifies a double-line border. A single thick-line border is not supported by the property that frame uses but you can obtain a thick line by drawing a double line then superimposing a single line via:

 1 frame 'b2:j19'

We also want double-line borders around the 3-by-3 grids, ie, below cells in rows 7 & 13, and on the right of columns 4 & 7 (D & G). *border* draws an ouside border for a single specified edge only; ‘b’=bottom and ‘r’=right:

 'b' 9 border 'b7:j7'

 'b' 9 border 'b13:j13'

 'r' 9 border 'd2:d19'

 'r' 9 border 'g2:g19'

Now let’s apply single-line borders to the individual cells; ie, below cells in rows 3, 5, 9, 11, 15, 17 and on the right of cells in columns B, C, E, F, H, I:

'b' border 'b3:j3'

'b' border 'b5:j5'

…

'b' border 'b17:j17'

and

'r' border 'b2:b19'

'r' border 'c2:c19'

…

'r' border 'i2:i19'

If you view the grid now you will see that Excel’s grid lines remain between the lower and upper portions of each Sudoku cell. We can eliminate those by setting a background colour for the entire grid. Colour 2 is the standard background colour so:

 2 bcolour 'b2:j19'

That completes the line drawing so let’s save the workbook before continuing. We could execute

 saveas 'book1' 'c:\apl\sudoku'

but Excel version 12 would save this file with a .xlsx suffix. It would be smarter, however, to anticipate some Excel macros and to save it with a .xlsm suffix, so let’s do this:

 saveas 'book1' 'c:\apl\sudoku.xlsm'

Be warned: if this file already exists, Excel will display a dialog box asking you to confirm that you want to overwrite the previous file. In many cases the dialog box will be obscured by the Apl session and all you will see, if you’re very observant, is a weakly flashing Excel session. It’s very easy to miss this cue. To avoid this problem, use ‘f’ (force the save) as a left argument to *saveas*. The alternative is ‘p’ (=prompt), the default. Thus:

 'f' saveas 'book1' 'c:\apl\sudoku.xlsm'

Excel could still prompt you for other reasons but this eliminates the commonest one. Note that su.Name has now changed:

su.Name

sudoku.xlsm

To finish off the grid, we now need to set row heights, fonts and justification for the top and bottom rows in each Sudoku cell. There is no toolkit function for row height so we will use the raw Excel command. The upper rows are 2×⍳9 and the lower rows are 1+ 2×⍳9. We will use row height 15 for the former and 30 for the latter:

#.XL.xl.Rows[2×⍳9].RowHeight←15

#.XL.xl.Rows[1+2×⍳9].RowHeight←30

xl.Rows is the Excel Rows Collection. (Similarly, you can set column widths via

#.XL.xl.Columns[2 3 4].ColumnWidth←30

or #.XL.xl.Columns[‘BCD’].ColumnWidth←30)

Beware that properties Height and Width exist too but are read-only.)

To left justify the upper cell in each Sudoku cell, do this for row 2:

 ljust 'b2:j2'

and repeat for rows 4, 6 … 18.

To centre each lower cell, do this for row 3:

 centre 'b3:j3'

and repeat for rows 5, 7, … 19.

At this point it’s worth illustrating the use of row-column co-ordinates instead of address strings. Compare these two code examples:

Using strings:

:For r :In 3 5 9 11 15 17

rr←⍕r

'b' border 'b',rr,':j',rr

 :EndFor

Using co-ordinates:

:For r :In 3 5 9 11 15 17

'b' border (r,2)(r,10)

 :EndFor

That completes the starting grid. Before we save it, let’s rename the sheet:

 rename '' 'Base'

An empty vector as the first argument value implies the active sheet. Referring to the active sheet is sometimes dangerous because we could have forgetten that we switched sheets, but it’s useful when you don’t want to think ‘what is the name/number of this sheet?’. It also leads to a shortcut in the code. If you check the Excel session, you will now see that the sheet has the name ‘Base’.

Now let’s save the workbook again. There is no ‘save’ function in the toolkit because all we need to do is:

 su.Save

If we hadn’t defined *su* earlier, an equivalent command is:

 #.XL.xl.ActiveWorkbook.Save

or else we could first define *su* via:

 su←setbook 'sudoku'

or su←setbook ''

In the second case, '' implies the active workbook.

## Defining a Puzzle

We want to anticipate a collection of puzzles – one per sheet – in this workbook so before we enter the initial values for a puzzle we’ll copy the Base sheet and rename it.

First, let’s see what sheets we have in this workbook:

sheets ''

 Base Sheet2 Sheet3

It’s not necessary, but we can remove the last two sheets since we don’t plan on using them:

deletesheet 2 3

We can specify sheets by number and/or number. Warning: you cannot delete all sheets.

Now we can copy the Base sheet:

copysheet 'base'

or copysheet 1

or copysheet ''

As in all cases where we specify worksheets, name (case independent) and number are both allowed.

Both commands copy the Base sheet to the end of the workbook (other options are possible). Thus:

sheets ''

 Base Base (2)

Now let’s rename the second sheet as ‘S1’ (our first Sudoku puzzle):

 rename 2 'S1'

Now it’s time to enter the initial values into the grid. For this, bring up the Excel window and enter the values in the **lower** portion of each Sudoku cell. A quick visual check: all the values should be centred; any left-justified numbers are in the upper portion. Here’s an example:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|   |   |   |   |   |   |   |   |   |
| 7 |   |   |   | 8 |   |   |   | 6 |
|   |   |   |   |   |   |   |   |   |
|   |   |   | 7 |   | 5 |   | 1 |   |
|   |   |   |   |   |   |   |   |   |
|   |   | 1 |   | 4 |   | 8 | 9 | 7 |
|   |   |   |   |   |   |   |   |   |
| 2 | 1 | 5 |   | 3 |   |   |   |   |
|   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |
|   |   |   | 8 |   |   | 7 | 6 | 2 |
|   |   |   |   |   |   |   |   |   |
|   | 3 |   |   | 2 |   | 6 |   |   |
|   |   |   |   |   |   |   |   |   |
|   | 8 |   | 9 |   | 7 |   |   |   |
|   |   |   |   |   |   |   |   |   |
|   |   |   |   | 3 |   |   |   | 5 |

So that we can later reset the grid to the starting point, we will now highlight these numbers in bold and set a background colour. This is actually a task that is better done in VBA but we’ll perform it in Apl as a learning experience. Here is code that will accomplish this (you can paste it into a Unicode Apl session):

 ∇ setGrid;grid;i;j;c;nu;allowed;r;c;rr;cc

[1]

[2] grid←read'b2:j19'

[3] rr←(2 4 6)(8 10 12)(14 16 18) ⍝rows comprising ea 3×3 block

[4] cc←(1 2 3)(4 5 6)(7 8 9) ⍝cols comprising each 3×3 block

[5] ⍝nb: grid[i;j] corresponds to Excel cell (i+1),(j+1)

[6]

[7] :For i :In 2×⍳9

[8] :If ∨/nu←grid[i;]≡¨⎕NULL

[9] r←(⌈i÷6)⊃rr ⍝minigrid rows

[10] :For j :In ⍳9

[11] :If grid[i;j]≡⎕NULL

[12] ⍝place allowed digits in upper cell

[13] c←(⌈j÷3)⊃cc ⍝minigrid cols

[14] allowed←1 0⍕(⍳9)~grid[i;],((18⍴0 1)/grid[;j]),,grid[r;c]

[15] allowed write i,j+1 ⍝Excel co-ords of upper cell

[16] :Else ⍝pretty up the lower cell

[17] bold 1+i,j ⍝Excel co-ords

[18] 24 bcolour 1+i,j ⍝mauve

[19] :EndIf

[20] :EndFor

[21] :EndIf

[22] :EndFor

 ∇

Here is the result of running *setGrid*:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|   | 2459 | 2349 | 123 |   | 1239 | 2345 | 2345 |   |
| **7** |   |   |   | **8** |   |   |   | **6** |
| 34689 | 2469 | 234689 |   | 69 |   | 234 |   | 34 |
|   |   |   | **7** |   | **5** |   | **1** |   |
| 356 | 256 |   | 236 |   | 236 |   |   |   |
|   |   | **1** |   | **4** |   | **8** | **9** | **7** |
|   |   |   | 46 |   | 469 | 49 | 48 | 489 |
| **2** | **1** | **5** |   | **3** |   |   |   |   |
| 34689 | 4679 | 346789 | 12456 | 15679 | 12469 | 13459 | 3458 | 13489 |
|   |   |   |   |   |   |   |   |   |
| 349 | 49 | 349 |   | 159 | 149 |   |   |   |
|   |   |   | **8** |   |   | **7** | **6** | **2** |
| 1459 |   | 479 | 145 |   | 148 |   | 478 | 1489 |
|   | **3** |   |   | **2** |   | **6** |   |   |
| 1456 |   | 246 |   | 156 |   | 1234 | 234 | 134 |
|   | **8** |   | **9** |   | **7** |   |   |   |
| 1469 | 24679 | 24679 | 146 |   | 1468 | 1249 | 2478 |   |
|   |   |   |   | **3** |   |   |   | **5** |

There is, of course, much more that *setGrid* could do but we have achieved the stated objective. Now let’s save the workbook again:

 su.Save

You can now try to solve the puzzle manually. Whenever you enter a value in a lower cell, be sure to remove the values from the upper cell and to remove the lower cell’s value from other upper cells in the same row, column or minigrid. This can be an Apl exercise for you – write a function that accepts a value and it’s grid co-ordinates; write the value in a lower cell then remove that value from affected upper cells. (You may contact the author if you want a function for this.)

As you proceed with the solution, you may make a mistake and want to return to the starting point. You can do this by interrogating the bold attribute in each cell and, if bold is not set, remove the values from both the lower and upper cells. For example, where Excel co-ordinates (i,j) denote a lower cell,

 #.XL.xl.Cells.(Item (i,j)).Font.Bold

returns 1 (set) or 0 (normal). If 0 is returned then you can execute

 clear ((i-1),j) (i,j)

to clear the upper and lower cells. After processing all the cells in the grid, execute *setGrid* again.

# Reading Values, Formulas, Text and Dates

The toolkit includes separate commands for reading values, formulas and text and a function for converting dates. These will be illustrated using the workbook loan.xlsx, which accompanies this documentation. If it’s missing or you have trouble opening it, part of it looks like this:

 Late Next Blended

Due Actual Payment Interest Fee Principal Amount Interest Payment

Date Date (1.5%) Reduction Owed Payment Eff 11/1/10

 8/1/2008 -25,000.00 25,000.00 625.00 3,486.68

11/1/2008 11/28/2008 1,000.00 625.00 9.38 365.62 24,634.38 615.86 3,435.69

2/1/2009 4/10/2009 1,000.00 615.86 0 384.14 24,250.24 606.26 3,382.11

This sheet contains a mixture of values, dates and formulas. First let’s open the workbook and read the values:

 startexcel ''

0

 open 'd:\apl\loan’

⍴r←read 'a2:i21'

20 9

 r

 [Null] [Null] [Null] [Null] Late [Null] [Null] Next Blended

 Due Actual Payment Interest Fee Principal Amount Interest Payment

 Date Date [Null] [Null] (1.5%) Reduction Owed Payment Eff 11/1/10

 [Null] 39661 ¯25000 [Null] [Null] [Null] 2.500000000E4 625 3486.68

 39753 39780 1000 625 9.38 365.62 2.463438000E4 615.86 3435.68

 39845 39913 1000 615.86 0 384.14 2.425024000E4 606.26 3382.11

 39934 39965 1500 606.26 0 893.74 2.335650000E4 583.91 3257.47

 40026 40049 1000 583.91 0 416.09 2.294041000E4 573.51 3199.43

 40118 40127 1000 573.51 0 426.49 2.251392000E4 562.85 3139.95

 40210 39860 4000 562.85 0 3437.15 1.907677000E4 476.92 2660.58

 40299 40333 1500 476.92 0 1023.08 1.805369000E4 451.34 2517.90

 40391 40380 4000 451.34 0 3548.66 1.450503000E4 362.63 2022.98

 40483 40483 2022.98 362.63 0 1660.35 1.284468000E4 321.12 2022.98

 40575 40574 2022.98 321.12 0 1701.86 1.114282000E4 278.57 2022.98

 40664 40666 2022.98 278.57 0 1744.41 9.398410000E3 234.96 2022.98

 40756 40763 2022.98 234.96 0 1788.02 7.610390000E3 190.26 2022.98

 40848 [Null] 2022.98 190.26 0 1832.72 5.777670000E3 144.44 2022.98

 40940 [Null] 2022.98 144.44 0 1878.54 3.899130000E3 97.48 2022.97

 41030 [Null] 2022.98 97.48 0 1925.5 1.973630000E3 49.34 2022.97

 41122 [Null] 2022.97 49.34 0 1973.63 ¯6.366462912E¯12 0 0

We have a mixed array which includes nulls. The possible surprise here is the date representation in the first two columns. Excel stores dates as integers, representing the number of days from January 1, 1900. The Value2 property of a cell (used in *read*) always returns the underlying value, which in this case is the day number. We therefore need to use *daynumToDate* to convert these numbers to formatted dates. *daynumToDate* allows four left arguments which control the format of the result. For example:

0 daynumToDate 39753 39780 ⍝nested array: (y m d day-of-week)

 2008 11 1 5 2008 11 28 4

1 daynumToDate 39753 39780 ⍝integer

20081101 20081128

2 daynumToDate 39753 39780 ⍝d/m/y

 01/11/2008 28/11/2008

3 daynumToDate 39753 39780 ⍝m/d/y

 11/01/2008 11/28/2008

In this example, we could replace the dates in the first two columns by:

r[;1 2]←2 daynumToDate r[;1 2]

*daynumToDate* converts positive integers only; other values are passed though unchanged. Thus:

7 2 ↑r

 [Null] [Null]

 Due Actual

 Date Date

 [Null] 01/08/2008

 01/11/2008 28/11/2008

 01/02/2009 10/04/2009

 01/05/2009 01/06/2009

To obtain formulas, use *readf*. For example, here’s the result for the first few lines (which are wrapped):

⍴rf←readf 'a2:i7'

6 9

 rf

 Late Next Blended

 Due Actual Payment Interest Fee Principal Amount Interest Payment

 Date Date (1.5%) Reduction Owed Payment Eff 11/1/10

 39661 =-B1 =B1 =ROUND($F$1\*G5,2) =ROUND(G5/((1-(1+$F$1)^-8)/$F$1),2)

 39753 39780 1000 =H5 =ROUND(D6\*0.015,2) =C6-(D6+E6) =G5-F6 =ROUND($F$1\*G6,2) =I$5\*G6/G$5

 39845 39913 1000 =H6 0 =C7-(D7+E7) =G6-F7 =ROUND($F$1\*G7,2) =I$5\*G7/G$5

 39934 39965 1500 =H7 0 =C8-(D8+E8) =G7-F8 =ROUND($F$1\*G8,2) =I$5\*G8/G$5

This shows formulas, where defined, or else values. However, every cell is returned as a string – a numeric value is converted to a string and an empty cell is returned as an empty string. Here’s proof:

rf[5;1]

 39753

⎕dr ⊃rf[5;1]

80

The third way to read cells is via *readt* which returns the Text property. For example:

⍴a←readt 'a2:i7'

6 9

a

 Late Next Blended

 Due Actual Payment Interest Fee Principal Amount Interest Payment

 Date Date (1.5%) Reduction Owed Payment Eff 11/1/10

 8/1/2008 -25,000.00 25,000.00 625.00 3,486.68

 11/1/2008 11/28/2008 1,000.00 625.00 9.38 365.62 24,634.38 615.86 3,435.69

 2/1/2009 4/10/2009 1,000.00 615.86 0 384.14 24,250.24 606.26 3,382.11

This returns exactly what Excel displays for each cell with the exception that all cells are left justified - Excel’s alignment properties are discarded. As with *readf*, all cells are returned as strings. This is a useful way of obtaining date strings and other formatted values.

## Reading a Dynamic Range

There are many cases where you want to read a range of cells and you know where the data starts but don’t know, without looking at the spreadsheet, where it ends. Sometimes the Used Range property of a sheet (all the defined cells – see *usedrange* in the Reference Manual) is what you want but it may return too much. To deal with this problem, the function *range* determines a range dynamically; given the starting point, it reads across and down (or up and left) until it reaches an empty cell. Here’s a sample Excel grid:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| **1** |  | Opponent | Score | Scorers |  |
| **2** | 13-Aug-11 | Newcastle | 0-0 |  |  |
| **3** | 16-Aug-11 | Udinese | 1-0 | Walcott |  |
| **4** | 20-Aug-11 | Liverpool | 0-2 |  |  |
| **5** | 24-Aug-11 | Udinese | 2-1 | Walcott, Persie |  |
| **6** |  |  |  |  |  |

*range* returns the row-column co-ordinates of the range based on the starting cell. For example:

 range ⊂'a1'

1 1 5 4 ⍝=A1:D5

Starting from A1, it reads across until it encounters a defined cell (D1) followed by an empty cell; D therefore marks the column limit. It then reads down until it encounters a defined cell (A5) followed by an empty cellned cell; 5 therefore marks the row limit. The range co-ordinates are therefore A1:D5, returned as (1 1)(5 4). The presence of data in F1 or A7, for example, would not affect the result. As table above grows over time, you can use the same code to read it:

 ⍴results←read range ⊂'a1'

5 4

You can pass the result of range to any function that expects a cell-range argument. You can also fix the row or column dimension and you can specify different scan directions. Please see the Reference Manual for more examples.

# Writing Values, Formulas and Dates

Here are two examples illustrating writing to a sheet, using *write* for data and *writef* for formulas and datestring. In both cases, the cell address in the right argument marks the top left corner of the target range:

 (?4 3⍴40) write 'a1'

('=sum(a1:a4)' '=sum(b1:b4)' '=sum(c1:c4)') writef 'a5'

And the proof:

read 'a1:c5'

27 23 37

 3 26 16

30 10 1

25 30 39

85 89 93

As the next example shows, you can mix data and formulas, and nulls and empty vectors are allowed too:

a←5 2⍴'Name' 'Score' 'Mary' 25 'Martha' 32 'Molly' 27 'Average' '=sum(b2:b4)/3'

a

 Name Score

 Mary 25

 Martha 32

 Molly 27

 Average =sum(b1:b3)/3

a write 'e1'

read 'e1:f5'

 Name Score

 Mary 25

 Martha 32

 Molly 27

 Average 28

In fact, we could have used *writef* instead of *write* in this case. *writef*, however, is essential if you write strings only. Repeating the first example in this section:

 (?4 3⍴40) write 'a1'

('=sum(a1:a4)' '=sum(b1:b4)' '=sum(c1:c4)') writef 'a5'

If you use *write* instead of *writef* the formulas are not evaluated:

('=sum(a1:a4)' '=sum(b1:b4)' '=sum(c1:c4)') write 'a5'

read 'a1:c5'

 6 31 19

 22 9 2

 28 28 38

 16 21 34

 =sum(a1:a4) =sum(b1:b4) =sum(c1:c4)

There are two ways to write dates. You can write day numbers (described in the previous section – Reading Values, Formulas, Text and Dates) or you can write text strings.

The function *dateToDaynum* converts integer dates in yyyymmdd format to Excel day numbers. You can then write the day numbers but you also need to set the date format (before or afterwards). For example:

dateToDaynum 2 1⍴20110920 20111101

40806

40848

(dateToDaynum 2 1⍴20110920 20111101) write 'a1'

readt 'a1:a2'

40806

40848

This result is what you will see if the format is either General or Number. You can now set the date format (you can do this before or after writing the day numbers):

'dd/mm/yyyy' format 'a1:a2'

readt 'a1:a2'

 20/09/2011

 01/11/2011

'm/d/yy' format 'a1:a2'

readt 'a1:a2'

 9/20/11

 11/1/11

In the paragraph above that discusses use of *writef* for formulas, the same principle applies to writing dates specified as text strings: if you write a set of date strings, you should use *writef* not *write*. These examples illustrate the problem:

a←2 4⍴ '13 Aug 2011' 'Newcastle' 0 0 '16 Aug' 'Udinese' 1 0

a

 13 Aug 2011 Newcastle 0 0

 16 Aug Udinese 1 0

a write ' a6'

read 'a6:d7'

40768 Newcastle 0 0

40771 Udinese 1 0

readt 'a6:d7'

 13-Aug-11 Newcastle 0 0

 16-Aug Udinese 1 0

Use of *read* proves that the date strings have been evaluated. The result of *readt* shows that the specified format has been preserved. Contrast the above examples with the ones below, where *write* does not evaluate the date strings but *writef* does:

(2 1⍴'Mar 3, 2011' '5 April 2011') write 'a8'

read 'a8:a9'

 Mar 3, 2011

 5 April 2011

(2 1⍴'Mar 3, 2011' '5 April 2011') writef 'a10'

read'a10:a11'

40605

40638

readt 'a10:a11'

 3-Mar-11

 5-Apr-11

It’s not always important to convert a date string to a day number, but doing this has some benefits – you can then re-format it, or perform date arithmetic, for example.

# xread and xwrite

These are two special functions that combine read and write functionality with starting and/or ending Excel, opening, creating or activating a workbook, and saving and closing a workbook. They also include comprehensive argument checking and error handling. Here are two examples:

1. Read the used range (ie, all the defined cells in the sheet, denoted by ‘\*’) from sheet 1 in d:\apl\loan. Excel is started if necessary, the workbook is opened or activated as necessary, and the ‘q’ flag terminates Excel, proved by executing *books* afterwards:

⍴r← 'q' xread 'd:\apl\loan' 1 '\*'

46 9

books''

Value Error: wb probably not defined - execute ∇startexcel

1. Write an array to the workbook d:\apl\portfolio, sheet ‘US’, starting at cell ‘E1’ and save it (parameter ‘s’); start a new Excel session (parameter’n’); leave Excel running (no ‘q’ parameter) and the workbook open:

us\_port←5 3⍴ 'JPM' 300 38.94 'LOW' 4000 22.968 'MON' 300 72.745 'ORCL' 900 23.338 'BTU' 100 56.17

'sn' xwrite 'd:\apl\portfolio' 'US' 'e1' us\_port

books ''

portfolio.xlsx d:\apl\portfolio.xlsx

# Plotting

The function *plot* is documented extensively in the Reference Manual, so please refer to that for more information. A few commands are demonstrated below.

The purpose of *plot* is to demonstrate that one can usefully build simple, experimental Excel plots in Apl. There are many plotting options not covered by *plot* but conceptually, at least, *plot* is easily extensible. In an application, however, rather than create a chart from scratch in Apl, it would be more efficient and less problematic to save a chart definition in a workbook and just update properties, such as labels and data, from Apl. Excel 12.0 supports 14 generic chart types and 73 individual chart types – plot also supports all these. Please see the ReferenceManual, Appendix B, for alist of chart types.

An important feature of *plot* is that it returns a chart object. Through this object, you can change chart attributes after you’ve created it – this, if you know how, is usually preferable to re-creating the chart. If you don’t know how to form the Excel command in Apl, then you can edit the chart in Excel.

Here are a few examples. The simplest command comprises ‘plot’ followed by a numeric vector:

 plot (⍳12)\*0.5

#.XL.xl.[\_Worksheet].[Shapes].[Shape].[\_Chart]

*plot* always uses the active sheet. It first writes the data (the right argument), starting at cell A1, then creates a chart in the same sheet. The result of plot in Apl is the chart object; the Excel result is this:

If you have two or more data series, the data must be presented as a matrix with a series in each column. Here’s an example of a chart with two series using stacked columns, and this time we’ll capture the result:

 c←'sc' plot ?8 2⍴40

Warning: If we execute the second plot command without changing the active sheet, both the data and the chart will overlay that from the first plot unless we specify in the plot argument where to locate both data and chart. In particular, the first chart will change if we specify different data for the second chart. The next instruction designates sheet 3 as the active sheet:

 s3 ← setsheet 3

By capturing the result, you can re-activate this sheet later by executing:

 s3.Activate

Usually we want to specify attributes such as a title and a legend. The way to structure the right argument is to specify a list of attribute name followed by the attribute’s value. For example:

c←'sc' plot 'data' (?8 2⍴40) 'title' 'Chocolate Sales in 2011' 'legend' ('Simplicity' 'Superchoc') 'ytitle Thousand' 'labelposn' 'F1' 'labels' (↑8 3⍴'JanFebMarAprMayJunJulAug')

with this result:

In this example we’ve added an array of labels and indicated where to place the labels in the sheet via the ‘labelposn’ parameter. For a summary of the available parameters, see the variable Plotspecs. For detailed information, please see the Reference Manual.

# Appendix

Here is a list of the toolkit functions grouped by category:

|  |  |  |
| --- | --- | --- |
| Starting and ending Excel | Startexcel | Initiate connection with Excel |
| Endexcel | Remove Excel objects from the workspace |
| Quitexcel | Terminate Excel and clean up the workspace |
| Dealing with Workbooks | Open | Open a saved or new workbook |
| Close | Close an open workbook |
| Saveas | Save a workbook under another name |
| books | List open workbooks |
| setbook | Designate an open workbook as the active workbook |
| Dealing with Worksheets | addsheet | Add sheet(s) |
| copysheet | Copy or move worksheet(s) |
| deletesheet | Delete worksheet(s) |
| rename | Rename worksheet(s) |
| sheets | List sheet names |
| setsheet | Designate a sheet as the active sheet |
| Reading Cells | read | Read values |
| readf | Read formulas  |
| readt | Read formatted text |
| usedrange | Read the Used Range |
| xread | See the Reference Manual |
| Writing Cells | write | Write data or mixed data, formulas and datestrings |
| writef | Write formulas and date strings |
| xwrite | See the Reference Manual |
| clear | Clear cell value |
| Formatting | format | Apply number or date format |
| ljust | Left justify |
| centre | Centre |
| rjust | Right justify |
| wrap | Set text wrapping |
| Decorative | bcolour | Set background colour |
| border | Set a specified outside border |
| border\_all | Set inside and outside borders |
| frame | Set all outside borders |
| Font Attributes | font | Set the font |
| fontcolour | Set font colour |
| fontsize | Set font size |
| bold | Set bold |
| Miscellaneous | range | Define a dynamic range – see the Reference Manual |
| cellrange | Return all cell co-ordinates comprising a range |
| lastcell | See the Reference Manual |
| lc | Convert string to lowercase |
| clean | Clean a data element before writing to a cell |
| rc | Convert address string to co-ordinates |
| rd | Convert address co-ordinates to a string |
| active | Return active book and sheet |
| dim | Return Excel worksheet dimensions |
| freeze | Freeze panes |
| help | Display ‘help’ information for each function |
| Plotting | plot | Create an Excel plot |
| charts | List names of all chart objects |
| Date Handling | daynumToDate | Convert Excel day numbers to dates |
| dateToDaynum | Convert dates to Excel day numbers |