Adventures in Groovy – Part 10: Validating Form Data

Introduction

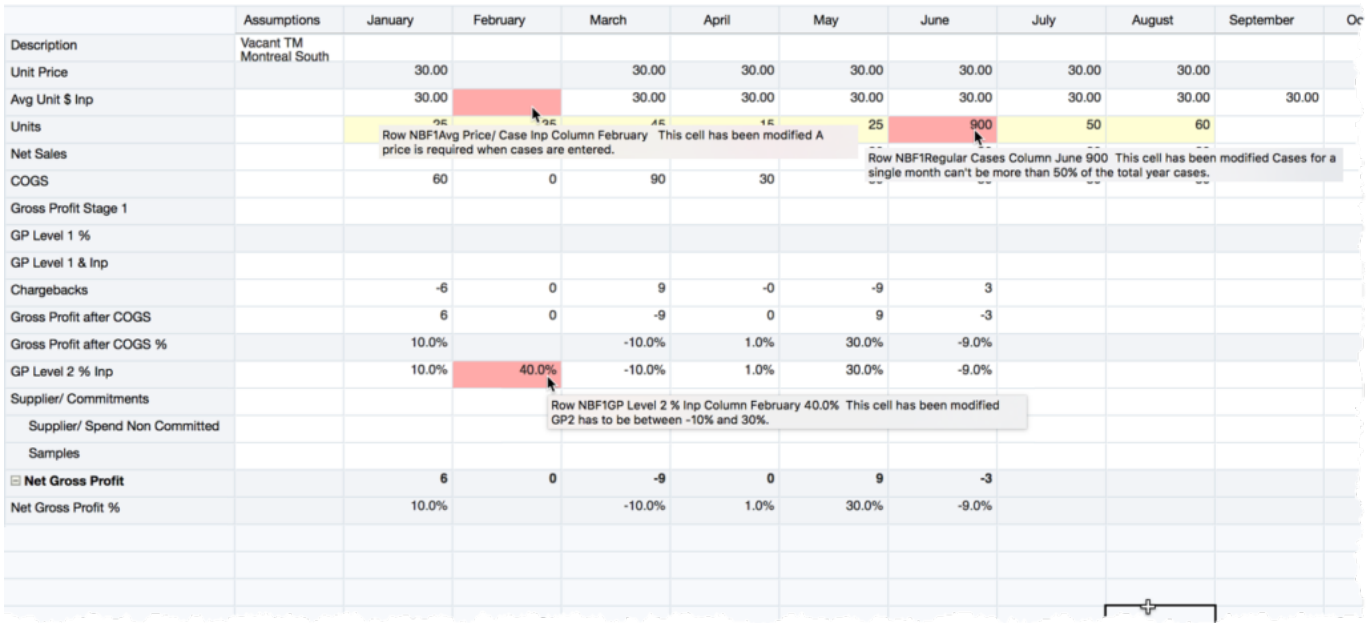
One of the huge frustrations I have with Planning is the fact that you haven't been able to stop a user from saving data that didn't validate since Smart View was released and Data Forms could be opened in Excel. Prior to that, the forms could be customized with JavaScript and the form save could be interrupted and cells flagged. Well, thanks to Groovy Calculations, it is back, and every bit as flexible.

Example

The following Data Form allows users to add new products and enter financial information. In this form, 3 rules exist.

1. The GP Level 2 % has to be between -10% and 30%.
2. The Regular Cases CANNOT have one month that is more than 50% of the total cases.
3. If Regular Cases is entered, a corresponding Average Price per Case is required.

When a user fills out the form and violates these rules, the cell background is changed and a tool tip is added. If violations exists, the form does NOT save any changes to the database. Before any changes can be committed, all errors have to be corrected. In this example, all 3 validation rules are violated and noted. If the user hovers the mouse over the cell with a violation, the tool tip is displayed with the row and column members, and an error message explains to the user what the issue is with the data that is entered.



	Assumptions	January	February	March	April	May	June	July	August	September	Oc
Description	Vacant TM Montreal South										
Unit Price		30.00		30.00	30.00	30.00	30.00	30.00	30.00		
Avg Unit $ Inp		30.00		30.00	30.00	30.00	30.00	30.00	30.00	30.00	
Units						25	900	50	60		
Net Sales											
COGS		60	0	90	30						
Gross Profit Stage 1											
GP Level 1 %											
GP Level 1 & Inp											
Chargebacks		-6	0	9	-0	-9	3				
Gross Profit after COGS		6	0	-9	0	9	-3				
Gross Profit after COGS %		10.0%		-10.0%	1.0%	30.0%	-9.0%				
GP Level 2 % Inp		10.0%	40.0%	-10.0%	1.0%	30.0%	-9.0%				
Supplier/ Commitments											
Supplier/ Spend Non Committed											
Samples											
Net Gross Profit		**6**	**0**	**-9**	**0**	**9**	**-3**				
Net Gross Profit %		10.0%		-10.0%	1.0%	30.0%	-9.0%				

The Code

The significance of this is huge, but the implementation is rather simple. It is probably be one of the more basic things created with a Groovy Calculation. Quite simply, to add a validation error and stop the form from saving, all that has to be done is to add a validation error to the cell.

cell.addValidationError(0xFF0000, "*Customer Error Message*",false)

This method accepts 3 parameters.

1. The first is the color you want the background to change to. This is the integer value of any color. Most people are familiar with the RGB code, and this can be retrieved in almost any image editor (even Windows Paint). There are many free options, like the free option at https://www.shodor.org/ to convert that to a value that can be interpreted in Groovy.
2. The second parameter is the error message to be displayed in the tool tip.
3. The third is optional, and defaults to false. False that it will identify the cell as an error and stop the form from saving.

This will likely becused in a grid iterator, which is how this example was constructed to get the screenshot above. If the grid iterator object is foreign to you, read Adventures in Groovy – Part 3: Acting On Edited Cells. The one function that is void from that article is the crossDimCell method. This acts like a cross dim (->) in a calculation. So, it references the POV of the cell and overrides the dimension of the member specified as a parameter. If multiple differences exist, separate the names with a comma.

```
def BackErrColor = 16755370 // Light Red
//Loop through the cells on the Regular Cases row
operation.grid.dataCellIterator('Regular_Cases','Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec').each {
  // Set a variable equal to all 12 months
    def CaseTotal = it.crossDimCell('Jan').data + it.crossDimCell('Feb').data + it.crossDimCell('Mar').data + it.crossDimCell('Apr').data + it.crossDimCell('May').data + it.crossDimCell('Jun').data + it.crossDimCell('Jul').data + it.crossDimCell('Aug').data + it.crossDimCell('Sep').data + it.crossDimCell('Oct').data + it.crossDimCell('Nov').data + it.crossDimCell('Dec').data
  // Check the cell value to see if it is larger than 50% of the total year
  if(it.data / CaseTotal > 0.5 )
     it.addValidationError(BackErrColor, "Cases for a single month can't be more than 50% of the total year cases.", false)
  // If cases are entered, make sure there is a corresponding price
                if(it.data != 0 && (it.crossDimCell("Avg_Price/Case_Inp").data == 0 || it.crossDimCell("Avg_Price/Case_Inp").data == '#Missing'))
it.crossDimCell("Avg_Price/Case_Inp").addValidationError(BackErrColor, "A price is required when cases are entered.", false)
}

// Loop throught the GP input cells and validate the % is in the valid range
operation.grid.dataCellIterator('GP_2_%_Inp','Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec').each {
```

```
  println "$it.MemberNames $it.data"
  if(it.data > 0.3 || it.data < -0.1 ) {
    it.addValidationError(BackErrColor, "GP2 has to be between
-10% and 30%.", false)
 }
}
```

Form Settings

The one gotcha is that this needs to run BEFORE SAVE. It makes sense, but I was expecting a similar result as validating a RTP when the Business Rule runs on save, so it took me a good night sleep to recognize that error in judgement.

Why This Is Significant

You may not thing this is a big deal because you can check this in a Business Rule after the data is saved and return an error message requesting the user to change it. However, the users are as busy, if not more busy, than you are. There are last minute changes that get slammed in at the end of a forecast or budget cycle. There is no design doc to go back to and say it is going to take longer and we need a change order. The CFO won’t accept that as an answer, so things get forgotten or missed. This example forces valid data (not necessarily accurate) to be entered, and all kinds of things can be checked to make sure human errors don’t turn into huge issues for financial reporting. Think if you had a product and forgot to put a price. You could be missing millions, and this type of proactive validation can prevent such things from happening. Little things like this reduce, or eliminate, fire drills later on in the budget cycle.

Conclusion

There is an infinite number of things that can be accomplished. Simple things like the above, to extremely

complex validation can be added. Think about ensuring allocated dollars are fully allocated(100%), forcing salaries to be in predefined pay bands for titles, and forcing the results of driver based planning to be within a logical margin.

If you have some examples, please share with the community by posting a comment below.