

# PyRFS User's Manual

© Keysight Technologies, Inc. 2016-2017

## Contents

Contents.....	2
Introduction .....	3
Quick start.....	4
Basic concepts.....	6
Constraint.....	6
Layout.....	7
Report .....	11
Directory structure of MQA project results .....	11
Tasks and examples .....	12
List constraints of a MQA project result .....	12
Sort values according to some conditions .....	13
Filter values according to some conditions .....	14
Format the value or name displayed in the table.....	14
Create a formula target based on other constraints .....	14
Add multiple tables in a sheet .....	14
Add multiple sheets in a book .....	15
Extract constraints from different MQA projects .....	15
Update existing Worksheet's contents.....	15
PyRFS API references .....	15
Classes.....	16
<i>Constraint, Condition, and Target.....</i>	<i>16</i>
<i>DataProvider and MQA_DataProvider .....</i>	<i>17</i>
<i>ReportItem, ReportSheet, ReportBook, and ReportTable .....</i>	<i>19</i>
<i>CellLayout, RightLayout, LeftLayout, DownLayout, and UpLayout .....</i>	<i>20</i>
Global functions .....	24
Errors.....	28
Appendix .....	29
Start the Python shell and editor.....	29
Attach PyRFS scripts into your MQA report.....	29

## Introduction

PyRFS is a Python library to create tables from some data sources and save them in an .xlsx file (Microsoft Office Open XML format for MS Excel since 2007). At present, the data source is limited to MQA results.

PyRFS is based on the concept of directly filling in a table with some data collections as a whole. It is no need for users to know the details of the data even the number of values of it. Only given the layout direction and some properties, then a table with sophisticate structure can be generated as their like.

A data collection in PyRFS is called a Constraint which limits the content of cells in a table. A Constraint could be a Condition or a Target. A Condition is a parameter or a variable based on which the values of a Target are calculated. For example in MQA, the Condition could be the rule group, rule, check, instance parameters, bias conditions, models, and so on. While, the Target can be the current Ids, lgs, the capacitance C, the S parameters, and so on. Users just add some conditions and targets into a Layout, then PyRFS can automatically determine value positions (cells in a table).

The difference between PyRFS and other table related libraries or programs is that for the former there is no need to know the cell position of a data at prior when putting it into a table, while for the later the precise cell position of it must be provided beforehand. For example, for the later, when users assign some values to cells, they should say something like *sheet ["B1:B3"] = [1, 2, 3]*, while for the former, they just say something like *right.add\_constraint (data)*, there is no need to explicitly specify the precise position like *["B1:B3"]*.

Besides the basic layout operation to arrange Constraints in a table, PyRFS also supports data sorting, selection, and grouping according to some conditions. Customized formulae and value display formats are also supported. PyRFS can also update the contents of an existing Excel Worksheet using tables it generated on Windows platform.

In summary, PyRFS provides the following functionalities:

- (1) Automatically generate tables and save them as Microsoft Excel XLSX files under Linux and Windows platforms
- (2) Support updating XLSX files under Windows
- (3) Automatically extract Constraints ( data collections) from MQA result directories and arrange them in tables
- (4) Automatically divide data into different tables, sheets, or files according to some conditions
- (5) Support value sorting and filtering for tables according to some conditions

- (6) Support displaying customized value formats and name formats of Constraints in tables
- (7) Support formulae contents calculated from other Constraints in tables

## Quick start

Fig.1 illustrates the basic usage of PyRFS.

Firstly, the library must be imported for the following usage as shown in line 2.

Then, line 7 creates a data provider with given *config* in line 6. The *config* specifies the directory where MQA results are located.

Line 8 queries conditions and targets with given *rule\_group*, *rule*, and *check*. And all conditions and targets can be obtained under the given conditions.

After querying the results, each target and condition can be obtained as shown in line 13 to line 18.

Then in line 21, a table is created with given name for putting the conditions and targets into it.

Now, a rightward layout is created for the table in line 22. A *RightLayout* means all constraints will be laid out from left to right by default. Line 23 to line 24 put *W*, *L* and *Idsat* in the table from left to right. Line 25 will put *T* over *Idsat* by the given parameters *offset*, *vdr\_direction* and *tgr\_direction*.

After the layout operation, a call of *fill\_in (table)* will fetch all values of *W*, *L*, *T*, and *Idsat*, and place them in the table.

The last work is saving the table in the given directory as line 32 said.

```

1  import os
2  import pyrfs as rfs
3
4  this_file_path = os.path.dirname(__file__)
5  project_path = os.path.join( this_file_path, "data", r"001_qa\qa_nchTT")
6  config = { 'result_dirs': project_path }
7  dp = rfs.create_data_provider(rfs.MQA_DataProvider, config)
8
9  dp.query( rule_group = "02_Model_Scalability",
10           rule = "Rule_4001",
11           check = "Check_01" )
12
13  Idsat = dp.targets["idsat"]
14
15  W = dp.conditions["w"]
16  L = dp.conditions["l"]
17
18  T = dp.conditions["t"]
19  T_count = T.get_num_values()
20
21  table = rfs.ReportTable( name = "demo_idsat_table")
22  right = rfs.RightLayout( table )
23  right.add_constraint( [W, L], offset = (1,0) )
24  right.add_constraint( Idsat, repeat = T_count )
25  right.add_constraint( T, ndr = 0, offset = (-1,-T_count),
26                      vdr_direction = rfs.LAYOUT_DIR_RIGHT,
27                      tgr_direction = rfs.LAYOUT_DIR_DOWN
28                      )
29
30  rfs.fill_in( table )
31
32  rfs.save(report = table, out_dir = r"C:\Temp")
33

```

Fig.1 Codes for the quick start example

The table generated by the above script is something like Fig.2 shown.

	A	B	C	D	E
1			-40	25	125
2	w	l	idsat	idsat	idsat
3	0.0000004	0.00000035	0.000341698	0.000315533	0.000269456
4	0.0000004	4.40624E-07	0.000312692	0.000277668	0.000227004
5	0.0000004	5.54713E-07	0.000278057	0.000238745	0.0001888
6	0.0000004	6.98342E-07	0.000241375	0.000201519	0.000155426
7	0.0000004	8.7916E-07	0.0002115	0.000171378	0.000128875
8	0.0000004	1.1068E-06	0.000180647	0.000142527	0.000105097
9	0.0000004	1.39338E-06	0.000151109	0.000117062	8.51373E-05
10	0.0000004	1.75416E-06	0.000124963	9.55012E-05	6.87146E-05
11	0.0000004	2.20835E-06	0.00010194	7.71621E-05	5.51157E-05
12	0.0000004	2.78015E-06	8.24712E-05	6.19969E-05	4.40587E-05
13	0.0000004	0.0000035	6.63666E-05	4.96407E-05	3.51499E-05
14	0.0000004	4.40624E-06	5.32173E-05	0.000039658	2.80074E-05
15	0.0000004	5.54713E-06	4.25687E-05	3.16348E-05	2.22977E-05
16	0.0000004	6.98342E-06	3.39917E-05	0.000025208	1.77418E-05
17	0.0000004	8.7916E-06	2.71087E-05	2.00715E-05	1.41111E-05
18	0.0000004	0.000011068	2.15992E-05	1.59726E-05	1.12199E-05
19	0.0000004	1.39338E-05	1.71975E-05	1.27055E-05	8.91915E-06
20	0.0000004	1.75416E-05	1.36856E-05	1.01034E-05	7.08898E-06
21	0.00010025	0.00000035	0.058638	0.0528885	0.0455723
22	0.00010025	4.40624E-07	0.0555827	0.0486431	0.040666
23	0.00010025	5.54713E-07	0.0511287	0.0433059	0.0351106
24	0.00010025	6.98342E-07	0.0458405	0.0376421	0.0286452

Fig.2 Table generated by the quick start example

## Basic concepts

### Constraint

A *Constraint* is a collection of values can be filled into and limit the contents of a table. Generally, a constraint has a name, a unit, a display name, and associated values. When a constraint is put into a table, there are three ranges to be determined. The first one called *ndr* (name display range) is cells the name occupied, the second one called *vdr* ( value display range ) is cells the values occupied, and the last one called *tgr* ( target range ) is cells in which the values are influenced by it. Fig.3 shows the three ranges for Constraint *W*. *ndr* of *W* is cell A4, *vdr* of *W* is cells [A5:A9], and *tgr* of *W* is cells [C5:H9].

There are two types of Constraints: Condition and Target. A condition is a constraint on which a target's values are dependent, and a target is one with values calculated from other conditions applied to it. For a target, *tgr* has no meaning, while for a condition, *tgr* means the target in the range will be dependent on it. For example, in Fig.3, *W*, *L* and *Model* (*TT*, *SS*, *FF*) are conditions, while *Vth* and *Idsat* are targets. *W*, *L* have the same *tgr* = [C5:H9], and *TT* has *tgr* = [C5:D9], *Vth* has *vdr* = [C5:C9], *Idsat* has *vdr* = [D5:D9] under the conditions *W*, *L* and *TT*. So the values in cells [C5:C9] will be *Vth* values under the corresponding conditions *W*, *L* and *TT*, and the values in cells [D5:D9] will be *Idsat* under the same conditions. In PyRFS, users are not required to explicitly calculate *Vth* and *Idsat* under the conditions, instead, they just construct a certain kind of *layout* (*RightLayout*, *LeftLayout*, *UpLayout*, or *DownLayout*) object, and add the conditions and targets into it by calling *layout.add\_constraint* (), PyRFS will automatically calculate the targets' values and put them in the correct cells as illustrated in the quick start example.

In PyRFS, there is no way to directly specify the three ranges, instead, they are automatically calculated with some given parameters to layout operations.

	A	B	C	D	E	F	G	H	
									1
									2
W is an instance parameter Constraint			TT		SS		FF		3
W.ndr	W	L	Vth	Idsat	Vth	Idsat	Vth	Idsat	4
	20	1							5
W.vdr	10	0.5							6
	5	0.2		1.2					7
	2	0.1							8
W.tgr	1	0.05							9

Fig.3 Constraint concept illustration

## Layout

In order to arrange constraints in a table correctly, a certain kind of layout must be setup for the table by constructing one object of *RightLayout*, *LeftLayout*, *UpLayout*, or *DownLayout* as illustrated by the quick start example (in which a *RightLayout* object was constructed).

A table is always located in a sheet with default starting location in cell A1. You can change the location when constructing a certain kind of layout by giving the parameter *location* a tuple of integers (*row*, *column*), for example, *right* = *RightLayout* (*table*, *location* = (3, 2)) will make the table put the next added constraint starting at cell B3.

There are four types of layout can be used in PyRFS to arrange constraints in a table. Each kind of layout has some default properties such as starting cell for the next *add\_constraint ()* operation and *vdr* and *tgr\_direction* for the added constraint.

Suppose the condition *W* has been added into the layout, the following figure shows the default position for the next *add\_constraint ()* operation (indicated by green symbol \*), the default *vdr* for the condition *W* (indicated by the blue cells), the default *tgr\_directions* of *W* (indicated by the orange symbol >, <, ^, and v). For example, a *RightLayout* will put the next added constraint in the right column of previously added *W*, and the rightward cells from *W* values will be influenced by them. In other words, it can be said that *RightLayout* will have *tgr* at rightward cells, while *LeftLayout* at leftward, *UpLayout* at upward, and *DownLayout* at downward cells.

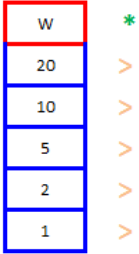
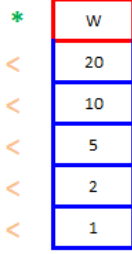
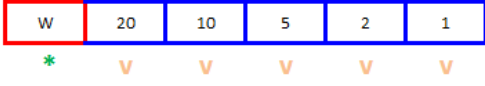
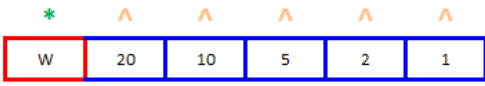
<i>RightLayout</i>	
<i>LeftLayout</i>	
<i>DownLayout</i>	
<i>UpLayout</i>	

Fig.4 Default *vdr*, *tgr\_direction*, and the next position for next *add\_constraint* operation



The default properties can be changed by passing special keyword arguments into the operation *add\_constraint*. The acceptable keyword arguments are listed in Table 1.

<b>Name</b>	<b>Description</b>	<b>Default Value</b>	<b>Constraints can be Applied to</b>
<b>offset</b>	<i>a tuple of integers (row, column) indicates offset to the current default layout position.</i>	(0,0)	Condition, Target
<b>ndr</b>	<i>name display range. Can only be 0. When it is 0, the name will not be displayed.</i>	N.A.	Condition, Target
<b>vdr</b>	<i>value display range. Can only be 0. When it is 0, the values will not be displayed.</i>	N.A.	Condition, Target
<b>vdr_direction</b>	<i>direction of the value display range</i>	LAYOUT_DIR_DOWN for RightLayout and LeftLayout;  LAYOUT_DIR_RIGHT for DownLayout and UpLayout	Condition, Target
<b>vdr_offset</b>	<i>offset of first value display range from its default position</i>	(0,0)	Condition, Target
<b>tgr_direction</b>	<i>direction of tgr from its first value</i>	LAYOUT_DIR_RIGHT for RightLayout;  LAYOUT_DIR_LEFT for LeftLayout;  LAYOUT_DIR_DOWN for DownLayout;  LAYOUT_DIR_UP for UpLayout	Condition
<b>tgr_offset</b>	<i>offset of tgr from its default position</i>	(0,0)	Condition
<b>row_span</b>	<i>the number of rows occupied by a value of a constraint</i>	1	Condition, Target
<b>col_span</b>	<i>the number of columns occupied by a value of a constraint</i>	1	Condition, Target
<b>max_rows</b>	<i>maximum number of rows occupied by the constraint. Only valid for downward and upward directions to limit the number of rows of tgr of Conditions or vdr of Targets.</i>	None	Condition, Target
<b>max_cols</b>	<i>maximum number of columns occupied by the constraint. Only valid for rightward and leftward directions to limit the number of columns of tgr of Conditions or vdr of Targets</i>	None	Condition, Target
<b>ndr_format</b>	<i>format string controlling how the name will be displayed.</i>	None	Condition, Target

Name	Description	Default Value	Constraints can be Applied to
<b>vdr_format</b>	format string controlling how the value will be displayed.	None	Condition, Target
<b>val_sort</b>	sort method when displaying its values. It should be as one of the following formats: val_sort = 'dec' or 'inc'; val_sort = { 1:val_sort_for_element_1, 2:val_sort_for_element_2, 3:val_sort_for_element_3 }	'inc'	Condition
<b>val_filter</b>	filter method for values. It should be set as one of the following formats: val_filter = ('range', (min, max)); val_filter = ('TT', 'SS', 'FF'); ( val_sort takes no effect for this case)  val_filter = bool_func; val_filter = { 1:val_filter_for_element_1, 2:val_filter_for_element_2, 3:val_filter_for_element_3 }	None	Condition
<b>repeat</b>	number of copies of the constraint to be laid out	1	Condition, Target
<b>val_group</b>	grouping method for the values from different elements. Set it as one of the following methods: val_group = 'zip'; val_group = 'loop'; val_group = ('loop', (3, 1, 2)). Here, the tuple (3, 1, 2) specifies the order of loop, in this case, element 3 will be the outer most loop variable, while element 2 will be the inner most loop variable.	'loop'	A list of Conditions
<b>hidden</b>	a tuple of indices indicates elements hidden from final report. The index value in it should be at least 1	None	A list of Conditions, A list of Targets
<b>gap</b>	number of empty rows or columns between two elements 0: the default, two elements will be arranged side by side; <0: two elements will be laid out in the same cell; n>0: two elements will be separated by gap number of rows or columns	0	A list of Conditions, A list of Targets

Table 1 Properties of *LayoutStyle* and keyword arguments can be passed into *add\_constraint* operation

The quick start example has shown some usages of the keyword arguments.

In line 23, when adding *W* and *L* in the layout, *offset* = (1, 0) makes *W* start at cell A2 as shown in Fig.2.

In line 24, when adding *Idsat* in the layout, *repeat* = *T\_count* makes *Idsat* laid out three times (the number of values of *T*) as shown in Fig.2.

In line 25, when adding condition  $T$  in the layout,  $offset = (-1, -T\_count)$  shift the current position for the *add\_constraint* operation one line up and three columns left.  $vdr\_direction = rfs.LAYOUT\_DIR\_RIGHT$  changes the default *vdr\_direction* from downward to rightward, and  $tgr\_direction = rfs.LAYOUT\_DIR\_DOWN$  changes the default *tgr\_direction* from rightward to downward. The combination of *vdr\_direction* and *tgr\_direction* makes *Idsat* below influenced by the condition  $T$ . And  $ndr = 0$  makes the name of  $T$  disappeared in the table.

## Report

A table in PyRFS is called *ReportTable*. PyRFS also provides *ReportSheet* to hold *ReportTables*, and *ReportBook* to hold *ReportSheets*. A *ReportBook* is corresponding to a *Workbook* in MS Excel, and a *ReportSheet* a *Worksheet*.

The functions *create\_sheets\_for\_book ()* and *create\_tables\_for\_sheet ()* can be used to add *ReportSheets* to a *ReportBook* and *ReportTables* to a *ReportSheet* respectively according to some conditions.

## Directory structure of MQA project results

At present, PyRFS can automatically extract Conditions and Targets from MQA project result directories. Generally, a valid result directory of a MQA project must satisfy that it or its child directory contains both the file *basicinfo.txt* and three sub-directories: *1\_summary*, *2\_detail*, and *3\_setup*. The structure of the sub-directory *2\_detail* is like:

*2\_detail/ [rule\_group\_category/] rule\_group/ rule/ [check\_group/] check/ io-files,*

in which *rule\_group\_category* and *check\_group* are optional. Each sub- directory in *2\_detail* is corresponding to a value of a built-in Condition: *rule\_group\_category*, *rule\_group*, *rule*, *check\_group*, or *check*. There is another built-in Condition *source* extracted from input file name, and its value could be a measurement site or a model. Figures 5 and 6 show the relationship between the built-in conditions and sub-directories.

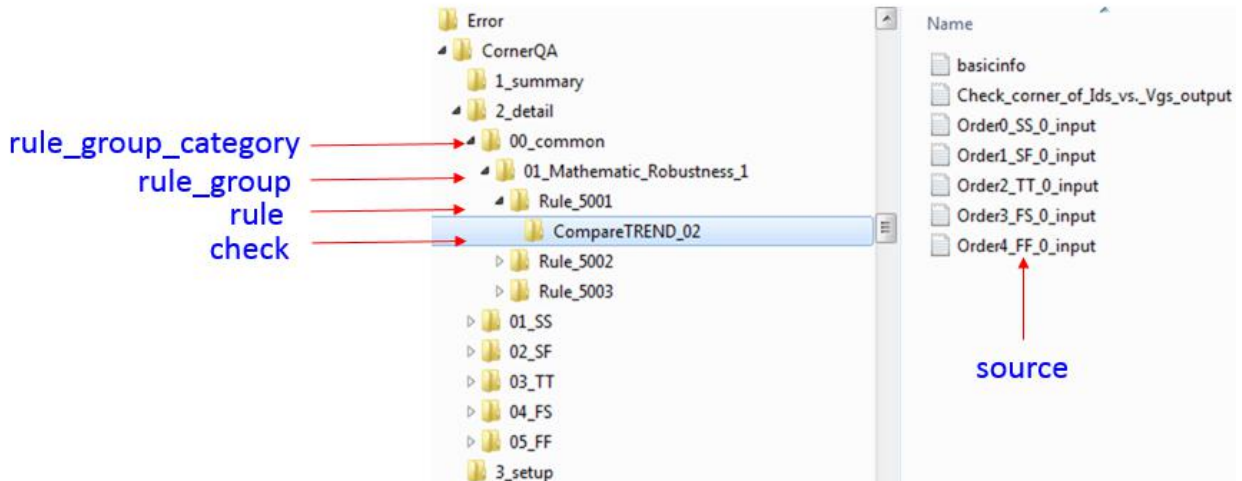


Fig.5 MQA result directory with rule group category

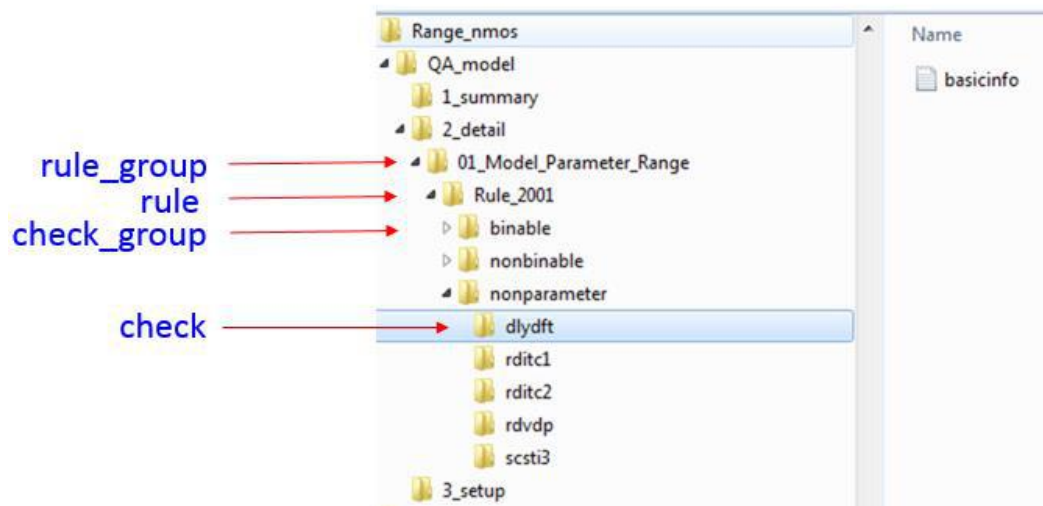


Fig.6 MQA result directory with check group

## Tasks and examples

### List constraints of a MQA project result

Users can find all conditions and targets of a MQA project using *DataProvider.query()* without any parameter and list them by calling *DataProvider.list\_conditions()* and *DataProvider.list\_targets()*. Fig.7 shows the usage of them and the parameter "detail=True" will list all values of the conditions

and sources of targets. Fig.8 is a snapshot of a partial output from the script in a Python shell. After inspecting all constraints this way, users can refine the results by passing parameters *project*, *rule\_group\_category*, *rule\_group*, *rule*, *check\_group*, or *check* into the method *DataProvider.query ()* as the quick start example shown.

```

1  import os
2  import pyrfs as rfs
3
4  this_file_path = os.path.dirname(__file__)
5  project_path = os.path.join( this_file_path, "data", r"001_qa\qa_nchTT")
6  config = { 'result_dirs': project_path }
7  dp = rfs.create_data_provider( rfs.MQA_DataProvider, config )
8
9  #
10 # query all conditions and targets with no param given
11 # You will see some warnings due to unsupported file format
12 #
13 dp.query()
14 dp.list_conditions( detail =True)
15 dp.list_targets( detail = True )

```

Fig.7 Script to list all conditions and targets of a MQA project result

```

w=[4e-07, 4.4e-07, 5.0357e-07, 6e-07, 6.33957e-07, 7.98105e-07, 1.00475e-06, 1.26491e-06, 1.59243e-06, 2e-06, 2.00475e-06, 2.52383e-06, 3.17731e-06, 4e-06,
5.0357e-06, 6.33957e-06, 7.98105e-06, 1.00475e-06, 1.091052632e-05, 1.26491e-05, 1.59243e-05, 2e-05, 2.00475e-05, 2.142105263e-05, 2.25889e-05, 2.52383e-05,
3.17731e-05, 3.193157895e-05, 4e-05, 4.034e-05, 4.244210526e-05, 4.47778e-05, 5.0325e-05, 5.0357e-05, 5.295263158e-05, 6.33957e-05, 6.346315789e-05, 6.69667e-05,
7.397368421e-05, 7.98105e-05, 8.028e-05, 8.448421053e-05, 8.91556e-05, 9.499473684e-05, 0.00010025, 0.000100475, 0.0001055052632, 0.000111344, 0.0001160157895,
0.00012022, 0.000126491, 0.0001265263158, 0.000133533, 0.0001370368421, 0.0001475473684, 0.000150175, 0.000155722, 0.0001580578947, 0.000159243, 0.00016016,
0.0001685684211, 0.000177911, 0.0001790789474, 0.00018009, 0.0001895894737, 0.0002001]

l=[3.5e-07, 3.85e-07, 4e-07, 4.2e-07, 4.40624e-07, 4.5e-07, 5.25e-07, 5.5e-07, 5.54713e-07, 6.98342e-07, 8.7916e-07, 1.1068e-06, 1.384736842e-06, 1.39338e-06,
1.75416e-06, 2e-06, 2.20835e-06, 2.316e-06, 2.419473684e-06, 2.53444e-06, 2.78015e-06, 3.454210526e-06, 3.5e-06, 4.282e-06, 4.40624e-06, 4.488947368e-06, 4.71889e-06,
5.265e-06, 5.523684211e-06, 5.54713e-06, 6.248e-06, 6.558421053e-06, 6.90333e-06, 6.98342e-06, 7.593157895e-06, 8.214e-06, 8.627894737e-06, 8.7916e-06, 9.08778e-06,
9.662631579e-06, 1.018e-05, 1.069736842e-05, 1.1068e-05, 1.12722e-05, 1.173210526e-05, 1.2146e-05, 1.276684211e-05, 1.34567e-05, 1.380157895e-05, 1.39338e-05,
1.4112e-05, 1.483631579e-05, 1.5095e-05, 1.56411e-05, 1.587105263e-05, 1.6078e-05, 1.690578947e-05, 1.75416e-05, 1.78256e-05, 1.794052632e-05, 1.8044e-05,
1.897526316e-05, 2.001e-05]

```

Fig.8a w and l conditions listed by script in Fig.7

```

weff_dc+w0:
model::Check_01[W0+weff>1e-7]-->Rule_6017[Check W0,weff]-->04_Key_Model_Parameter_Check[Key Model Parameter Check]-->qa_nchTT[qa_nchTT]-->MQA[MQA]
model::Check_02[W0+weff<0]-->Rule_6017[Check W0,weff]-->04_Key_Model_Parameter_Check[Key Model Parameter Check]-->qa_nchTT[qa_nchTT]-->MQA[MQA]

y21_i:
model::Check_16[Check Y21_I]-->Rule_4016[Check Y parameter]-->02_Model_Scalability[Model Scalability]-->qa_nchTT[qa_nchTT]-->MQA[MQA]

vth_sat:
model::Check_01[Plot Trend]-->Rule_4005[Check Vth_sat vs. L]-->02_Model_Scalability[Model Scalability]-->qa_nchTT[qa_nchTT]-->MQA[MQA]
model::Check_01[Plot Trend]-->Rule_4007[Check Vth_sat vs. W]-->02_Model_Scalability[Model Scalability]-->qa_nchTT[qa_nchTT]-->MQA[MQA]
model::Check_01[Check Trend]-->Rule_4009[Check Vth_sat vs. T]-->02_Model_Scalability[Model Scalability]-->qa_nchTT[qa_nchTT]-->MQA[MQA]
model::Check_02[Check Kink]-->Rule_4009[Check Vth_sat vs. T]-->02_Model_Scalability[Model Scalability]-->qa_nchTT[qa_nchTT]-->MQA[MQA]

```

Fig.8b some targets listed by script in Fig.7

The example demo/demo\_01\_\_query\_\_MQA\_\_result.py is the full script for playing with this task.

## Sort values according to some conditions

PyRFS supports value sorting by Condition objects. Users can pass *val\_sort*--> argument to add\_constraint operation to do the sorting job. See demo/ demo\_\_06\_\_val\_sort\_\_table.py for example.

### Filter values according to some conditions

PyRFS supports value filtering by Condition objects. Users can pass *val\_filter* argument to *add\_constraint* operation to do the filtering job. See demo/ demo\_\_07\_\_val\_filter\_\_table.py for example.

### Format the value or name displayed in the table

PyRFS supports changing the display format of values and names of Constraints. Users can use *ndr\_format* to format the name displayed and *vdr\_format* to format the values displayed in the table when doing *add\_constraint* operation. *vdr\_format* and *ndr\_format* must be a valid string expression as those described in the following creating formula target. Also, a special slice type formula pattern is also support for the *vdr* indexing to a single *Target*:

"(\\$)\[s\*(\.[0-9]\*))s\*\]". Here  $\$[.]$  refers to current *vdr*, and  $\$[i]$  refers to *i*-th *vdr*.

Examples include:

demo/demo\_\_04\_\_WL\_in\_same\_column\_\_table.py

demo/demo\_\_13\_\_ndr\_vdr\_format.py

demo/demo\_\_15\_\_np\_compare.py

### Create a formula target based on other constraints

PyRFS supports add a formula into a table by *add\_formula* operation. The formula is a string expression for what data will be extracted from constraints added. The string must be a valid Python expression after replacing the constraint patterns in it. Two types of constraint patterns are supported:

1. "(\\$)\[.([v|u|n])]" for one constraint with '\$' refers to the constraint. 'v' is the value, 'u' is the unit, 'n' is the name.
2. "(\\$([1-9]))([.([v|u|n])?)?" for a list of constraints, with '\$i' refers to the (i-1)-th element of the constraint list.

Examples include:

demo/demo\_\_05\_\_formula\_\_table.py

### Add multiple tables in a sheet

PyRFS supports multiple tables in a sheet. You can directly append a table into *ReportSheet's* property *tables* or call *create\_tables\_for\_sheet* to let PyRFS automatically generate a series of tables according to some conditions.

Examples include:

demo/ demo\_\_09\_\_table\_group.py

demo/demo\_\_15\_\_np\_compare.py

### Add multiple sheets in a book

PyRFS supports multiple sheets in a book. You can directly append a sheet into *ReportBook*'s property *sheets* or call *create\_sheets\_for\_table* to let PyRFS automatically generate a series of sheets according to some conditions.

Examples include:

demo/ demo\_\_10\_\_sheet\_group.py

### Extract constraints from different MQA projects

If two MQA projects have the same rule settings, then users can extract constraints from the two projects to build a compare table from it. Users just pass two result directories into *create\_data\_provider* function to let it happen. See demo/ demo\_\_14\_\_vth\_from\_two\_projects.py for example.

### Update existing Worksheet's contents

PyRFS supports updating existing Excel files using ReportTable contents by calling *update\_sheet* function. See demo/demo\_\_11\_\_update\_sheet.py for example.

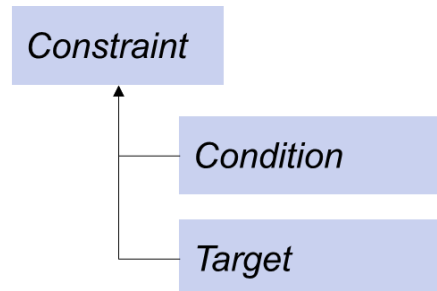
## PyRFS API references

In this reference, only classes and functions users can call are listed. Calling other methods/functions or properties not listed in here are not recommended and may cause undefined behaviors of the program.

## Classes

### *Constraint, Condition, and Target*

Hierarchy structure is shown in the following figure. It's no need for users to use them directly, just query them from a data provider.



#### ***class Constraint***

*methods:*

```
def __init__ (self, name=""):
    """
    Initialize an instance of a Constraint object
    @param name: name of the object
    """
```

*properties:*

```
self.name:           # name of the object
self.display_name:   # name to be displayed in a table
self.unit:           # unit of the object
```

#### ***class Condition***

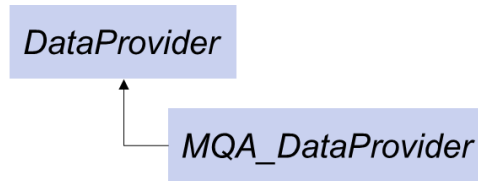
*methods:*

```
def get_num_values (self):
    """ Return the number of values in this condition """
```



### *DataProvider and MQA\_DataProvider*

Hierarchy structure is shown in the following figure. Users can use the function `create_data_provider ()` to create an instance of *MQA\_DataProvider* .



#### ***class DataProvider***

*methods:*

```
def __init__ (self, name=""):
```

```
    """
```

```
    Initialize an instance of this class
```

```
    @param name: name of the object
```

```
    """
```

```
def query (self, **kwargs):
```

```
    """
```

```
    query data based on input args.
```

```
    The data are stored in properties conditions and targets
```

```
    @param kwargs: other keyword args for the query
```

```
    """
```

*properties:*

```
self.targets:                # a dict mapping names to Target objects in current query
```

```
self.conditions:            # a dict mapping names to Condition objects in current query
```

#### ***class MQA\_DataProvider***

methods:

```
def __init__ (self, result_dirs, ignore_case=True, compact_mode = True):
```

```
    """
```

*Initialize an instance of this class*

*@param result\_dirs: result directories of MQA projects*

*@param ignore\_case: True when search the constraint name by case-insensitive  
method. False by case-sensitive method*

*@param compact\_mode: If True, any condition with only one value will be ignored!*

```
    """
```

```
def query (self, **kwargs):
```

```
    """
```

*Query the targets from the result dir*

*@param kwargs: keyword arguments for the node. Acceptable arguments include*

*project: a str or list of str specify the project(s), default is None.*

*rule\_group\_category: a str or list of str specify the rule group category(s), default  
is None.*

*rule\_group: a str or list of str specify the rule group(s), default is None.*

*rule: a str or list of str specify the rule(s), default is None.*

*check\_group: a str or list of str specify the check group(s), default is None.*

*check: a str or list of str specify the check(s), default is None*

*If any of the above params is None, then all of the values of the param in the  
above will be selected.*

```
    """
```

```
def list_targets (self, print_it = True, detail=False):
```

```
    """
```

*Return a list of all target names in this provider*

*@param print\_it: if True, then display the targets in Python shell*

*@param detail: if True and print\_it is also True, then print all the directory information of the targets*

"""

*def list\_conditions(self, print\_it = True, detail=False):*

"""

*Return a list of all condition names in this provider*

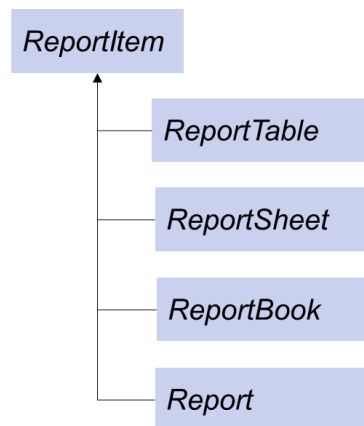
*@param print\_it: if True, then display the conditions in Python shell*

*@param detail: if True and print\_it is also True, then print all values of each condition*

"""

*ReportItem, ReportSheet, ReportBook, and ReportTable*

Hierarchy structure between them is shown in the following figure.



**class ReportItem**

methods:

*def \_\_init\_\_ (self, parent=None, name=""):*

"""

*Initialize an instance of this class*

*@param parent: report item containing this one. No need to set it by users explicitly.*

*@param name: name of this object*

"""

*properties:*

*self.name:       # name for this item*

*self.elements:   # a list of report items it contains*

***class ReportBook***

*properties:*

*self.sheets:     # a list of ReportSheets it contains*

***class ReportSheet***

*properties:*

*self.tables:     # a list of ReportTables it contains*

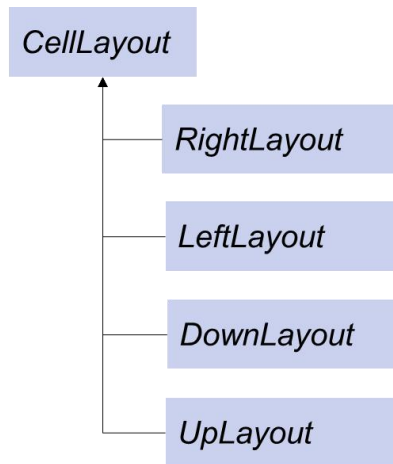
***class Report***

*properties:*

*self.books:       # a list of ReportBooks it contains*

*CellLayout, RightLayout, LeftLayout, DownLayout, and UpLayout*

Hierarchy structure is shown in the following figure. Users should create an instance of *RightLayout, LeftLayout, DownLayout, or UpLayout* to arrange constraints in tables.



***class CellLayout***

*methods:*

***def \_\_init\_\_*** (*self*, *table=None*, *\*\*layout\_props*):

*"""*

*Initialize an instance of it.*

*@param table: a ReportTable object contains cells*

*@param layout\_props: keyword arguments for layout properties.*

*@raise RfsTypeError: if report is not a ReportTable object*

*"""*

***def add\_offset*** (*self*, *row\_offset = 0*, *column\_offset = 0*):

*"""*

*Move the default position for the next layout operation*

*@param row\_offset: number of rows to move the default position*

*of the next layout operation*

*@param column\_offset: number of columns to move the default position*

*of the next layout operation*

*"""*

***def add\_layout*** (*self*, *layout*, *offset = None*):

"""

*Add a sub layout to this layout*

*The added layout will not change this layout's default position for next add\_xxx operations*

*The added layout's style such as location and offset will be ignored*

*@param layout: sub layout object added to this layout*

*@param offset: offset (row, column) from this layout.*

*@return the sub layout object*

"""

**def add\_constraint** (self, constraint, \*\*layout\_props):

"""

*Add a general constraint of a list of constraints into this layout.*

*@param constraint: a constraint or a list of constraints to be added*

*@param layout\_props: special layout properties for the constraint*

*In addition to the keywords in Table 1, it can also*

*contain 'val\_reference' keyword:*

*'val\_reference': specify the constraints returned by*

*last\_constraint\_references () to be referred by 'vdr\_format'*

*to setup the formula for value display. In this case, the "\$1" will*

*always be the constraint to be added.*

*@return this layout object (self)*

*@raise RfsValueError if constraint is an invalid Constraint object*

"""

**def add\_data**(self, name= "data", data=None, \*\*layout\_props):

"""

*Add a free data set into the layout.*

*@param name: name of the data*

*@param data: a single value or an iterable for values to be added into the layout*

```

@param layout_props: special keyword layout properties for the constraint

@return this layout object (self)

"""

def add_formula(self, name = 'formula', constraints = None, formula = None, **layout_props):
    """
    Add a formula into the layout.

    @param name: name for this formula

    @param constraints: constraints from which the formula get the data.

        If constraints is None, then will use all constraints in current layout
        when calling this method, and the first constraint will be the first one added
        in this layout, and the last constraint will be the last one added

    @param formula: a string expression like "$1+sin($2)+cos($3)"
        with '$i' refers to the (i-1)-th element of the constraints.

    @param layout_props: special layout properties for the constraint

    @return this layout object (self)

    """

def last_constraint_references (self, amount=1):
    """
    Return a list of the last amount of references to constraints in this layout.
    References in sub layout or parent layout are excluded.

    The returned references can be assigned to "constraints" in add_formula() to setup the
    formula or assigned to 'val_reference' in add_constraint() as list of constraints to
    setup 'vdr_format'.

    @param amount: number of references to be returned.

        If it is larger than the total number of references in current layout,
        then only the total number of them will be returned.

    """

```

## Global functions

```
def create_data_provider (dp_class, config):
```

```
    """
```

*Generate and return an instance of a subclass of DataProvider.*

*@param dp\_class: name of the subclass of DataProvider, such as MQA\_DataProvider*

*@param config: keyword args passed to \_\_init\_\_ ( ) method of the dp\_class*

```
    """
```

```
def create_tables_for_sheet ( sheet,
```

```
                                condition = None,
```

```
                                val_sort = None,
```

```
                                val_filter = None,
```

```
                                val_group = None):
```

```
    """
```

*Create tables for the sheet according to the condition.*

*@param sheet: a ReportSheet into which tables will be created*

*@param condition: a Condition object or a list of Condition objects, with each value of it  
a table will be created*

*@param val\_sort: sort method for the condition values*

*@param val\_filter: filter method for the condition values*

*@param val\_group: value grouping method for the condition*

```
    """
```

```
def create_sheets_for_book ( book,
```

```
                                condition = None,
```

```
                                val_sort = None,
```



```

        val_filter = None,
        val_group = None):
    """
    Create sheets for the book according to the condition.
    @param book: a ReportBook into which sheets will be created
    @param condition: a Condition or a list of Condition objects, with each value of it
                    a sheet will be created
    @param val_sort: sort method for the condition values
    @param val_filter: filter method for the condition values
    @param val_group: value grouping method for the condition
    """

def create_books_for_report ( report,
                             condition = None,
                             val_sort = None,
                             val_filter = None,
                             val_group = None):
    """
    Create books for the report according to the condition.
    @param report: a Report into which books will be created
    @param condition: a Condition or a list of Condition objects, with each value of it
                    a book will be created
    @param val_sort: sort method for the condition values
    @param val_filter: filter method for the condition values
    @param val_group: value grouping method for the condition
    """

def fill_in ( table):
    """

```

*Fill in the table with constraints' data in it.*

*It must be called before the table can be saved.*

"""

```
def save ( report=None, out_dir="", format='xlsx', overwrite=True):
```

"""

*Save the report into out\_dir with the given format.*

*@param report: report to be saved*

*@param out\_dir: diretory to save the report*

*@param format: 'xlsx' for .xlsx file format , and 'csv' for .csv file format*

*@param overwrite: If True, overwrite the file if the file is already exist.*

"""

```
def update_sheet (src_table=None,
```

```
                  src_range=None,
```

```
                  dest_dir=None,
```

```
                  dest_book_name=None,
```

```
                  dest_sheet_name=None,
```

```
                  dest_begin=None,
```

```
                  use_excel = True ):
```

"""

*Update the content of a MS Excel Worksheet with the given table contents*

*@param src\_table: table to be copied to sheet*

*@param src\_range: a cell range of "A1:C2" type string*

*specified the range of the table to be copied*

*@param dest\_dir: directory of the Excel Workbook to be updated*

*@param dest\_book\_name: name of the Excel Workbook to be updated*

*@param dest\_sheet\_name: name of the Excel Worksheet to be updated*

*@param dest\_begin: a cell index of "A1" type string specifying*

```

        the beginning of the range of the sheet to be updated

    @param use_excel: If True, then try best to call Excel to update the file.

        If Excel is not available, then charts in the Excel Workbook
        will be lost at present.

    @return the backup workbook file name
    """

def get_config()
    """

    Return the global config object for users to set some global PyRFS configurations.

    The available configurations and the default values include:

    # maximum allowed number of rows
    max_row_number = 10000

    # maximum allowed number of columns
    max_column_number = 1000

    # relative tolerance when two float numbers are evaluated to be equal. For example,
    # if  $|(a-b)/\max(|a|, |b|)| < \text{tolerance}$ , then  $a == b$  is True
    tolerance = 1.0e-10

    # control what message will be shown
    # 0: no message will be shown
    # 1: only warning message
    # 2: informative message + warning message
    verbosity_level = 1
    """

```

## Errors

In addition to common Python exceptions, PyRFS provides its own exceptions when an illegal operation occurred in its classes or functions. Those exceptions include:

***RfsTypeError***: argument type is incorrect

***RfsValueError***: argument value is incorrect

***RfsAttributeError***: there is no such attribute

***RfsIndexOutOfBoundsError***: index is out of bound

***RfsUnsupportedFileError***: the file format is unsupported now

***RfsNotImplementedError***: the function is not implemented yet

***RfsAssignmentError***: assign value to a read-only param

***RfsAccessError***: illegal access a method or property

***RfsRuntimeError***: unexpected run time error

***RfsInvalidMQAResultDir***: the directory is an invalid MQA result directory

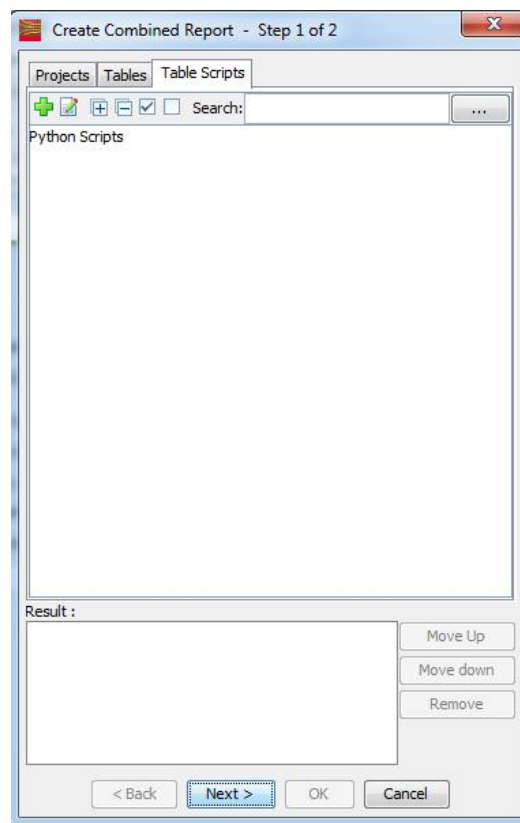
## Appendix


### Launch the Python shell and editor


You can find mpython.bat (for Windows) or mpython.sh (for Linux) in \$MQAHOME/bin directory (where \$MQAHOME is the directory MQA installed). Just run it, then python shell will launch up and ready for you to write and test PyRFS enabled Python scripts.

### Attach PyRFS scripts into your MQA reports

In Lib Explorer, choose 'Create Combined Report', you can find the sheet 'Table Scripts' in the pop-up panel as shown in the following figure.



1. Click the add button  to open a file selection dialog to select the python scripts attached to your report. When you generate the report, the selected scripts will run automatically for you to generate or update your tables.

2. Click the edit button  to open Python editor to modify the selected scripts when needed. After make the changes and close the editor, you can return to the report creation panel and continue the report creation work.