Adventures in Groovy – Part 25: Groovy Functions

Building on the previous post and in the spirit of reusing code, this will expand on the concept by walking through the creation and use of functions in Groovy calculations. Functions have huge value in reducing your Groovy calculations and streamlining development.

Functions allow access to processes that either do some action(s) or return a value to be used. There are hundreds of cases I can think of where this is valuable. Here are a few to get you started.

1. Getting the form POV, or a dimension in the POV
2. Creating and executing data maps (automatically using the form POV, for example)
3. Identifying if the account should be converted to a different currency (headcount will never be converted)
4. Logging repetative messages to the job console

Share your ideas with the community by posting a comment. I am sure your thoughts will be valuable to the community!

These functions can be kept in a script that is reusable as discussed in Part 24. Or, they can be created in individual Groovy calculations if they are unique to a specific calculation.

The Anatomy Of A Function

Functions can be dissected into three pieces.

1. A name (required)
2. The parameters (not required)
3. The logic (required)

The construction is below.

```
def name(parameter1, parameter2, ...)
{
  logic
}
```

The name is simple. This is a reference to the function and the string you use to call it. I would suggest making it meaningful, but not terribly long. Typing long names gets annoying after a while! Also, come up with a consistent case strategy. I normally capitalize the first letter in all but the first word (getSomethingGood).

The function can have parameters. Unless it is just doing some generic action, it will need something. It can be a numeric value, a string, a boolean value, or really any other type of object, like collections. Parameters are separated by commas and can have default values.

The logic is what is inside the function and the whole purpose to have a function. Functions can return values, execute some action, or both.

Function Examples

Getting the POV

A lot of functions need members from the POV. Sometimes surrounding them with quotes causes problems in the PBCS functions so this gives an option to include or exclude them. This also will return the member as a list for functions that require a list, like data maps.

The second two parameters are not required unless quotes or a list is needed as the return value.

```
def getPovMbr(dimension, boolean quotes = false, boolean
returnAsList = false) {
  if(returnAsList == true)
    // tokenize will split the value based on a delimiter and
convert the string to
```

```
    // a collection.  Since this is one value and we just need
the one value converted
    // to a collection, I used ### as the delimiter since it
will never occur in any
    // dimension name.
    return dimension.tokenize('###')
  else if(quotes == true)
                                return        '"'        +
operation.grid.pov.find{it.dimName==dimension}.essbaseMbrName
+ '"'
  else
                                                       return
operation.grid.pov.find{it.dimName==dimension}.essbaseMbrName
}
```

Assuming the Plan is selected in the POV,

- getPovMbr(“Scenario”) will return OEP_Plan
- getPovMbr(“Scenario”,true) will return “OEP_Plan”
- getPovMbr(“Scenario”,false,true) will return [OEP_Plan]

Print To The Log

Monitoring processing times of actions inside a calculation is helpful to diagnose issues and manage performance. This function will accept a message and return the duration from the previous execution of the function.

```
def startTime = currentTimeMillis()
def procTime = currentTimeMillis()
def elapsed=(currentTimeMillis()-startTime)/1000

def printPerformance(message, boolean printTotalTime = false)
{
  // Gets the elapsed time
  elapsed=(currentTimeMillis()-procTime)/1000
  // Sets the new start time
  procTime = currentTimeMillis()
  // Print to log
                                                      println
"*****************************************************"
  println "$message = $elapsed secs"
  if (printTotalTime = true)
      println "Total Time = " + (currentTimeMillis()-startTime)/1000 + " secs"
  println "****************************************
}
```

printPerformance("The Data Map processed",true) would write the following message in the log.

```
****************************************************
The Data Map Processed = .204 secs
Total Time = 1.44 secs
****************************************************
```

This can obviously be customized to whatever you want to show. As it is something used often, having a function saves a lot of time, reduces the non-business related code in the rule, and makes reading the business rule more digestible.

Rather than repeat all this,

```
time elapsed=(currentTimeMillis()-procTime)/1000
procTime = currentTimeMillis()
println "****************************************************"
println "$message = $elapsed secs"
println "Total Time = " + (currentTimeMillis()-startTime)/1000 + " secs"
println "****************************************
```

you simply call the function.

```
printPerformance("The Data Map processed",true)
```

Convert Account To USD

Groovy calculations don't need to run Essbase calculations to execute business logic. The logic can be executed in Groovy. For a presentation in Orlando, I wanted to prove this. I replicated a calculation that calculates revenue, gross profit, and margins. It also needs to produce converted currencies. As you know, accounts like units, headcount, and rates don’t get converted even if the local currency is something other than USD. In the example I showed at KScope, a gridbuilder was used and every account needed to be identified as an account that would be converted, or not converted. The following function returns a true/false for an account based on whether that account has a UDA of IgnoreCurrencyConversion.

```
boolean convertCurrency(def account)
{
                   Dimension         AccountDim          =
operation.application.getDimension("Account")
  Member AccountMbr = AccountDim.getMember(account)
  def memberProps = AccountMbr.toMap()
if(memberProps['UDA'].toString().contains('IgnoreCurrencyConve
rsion'))
    return false
  else
    return true
}
```

convertCurrency(“Units”) would return a false
convertCurrency(“Revenue”) would return a true

In the code, as it is looping through the accounts, it applies the conversion only if it needs to.

```
if(convertCurrency(account)){
  sValuesUSD.add(currencyRates[cMonth].toString().toDouble() *
setValue)
  addcellsUSD << currencyRates[cMonth].toString().toDouble() *
setValue
}
else
{
  sValuesUSD.add(setValue)
  addcellsUSD << setValue
}
```

The full example of this is near the end of the *ePBCS*

Gridbuilder Deep Dive – Last Minute KScope Souvenirs in my Kscope Wrap Up.

Conclusion

There are significant benefits to functions. As your growth in this space grows, you will likely develop more of these function and reuse them. Got an idea? Share it by posting a comment!