Paper AP-108-2024

Macro Variable Arrays Made Easy with `macroArray` SAS® Package

Bartosz Jabłoński - yabwon / Warsaw University of Technology

ABSTRACT

A macro variable array is a jargon term for a list of macro variables with a common prefix and numerical suffixes. Macro arrays are valued by advanced SAS® programmers and often used as "driving" lists, allowing sequential metadata for complex or iterative programs. Use of macro arrays requires advanced macro programming techniques based on indirect reference (aka, using multiple ampersands `&&`), which may intimidate less experienced programmers.

The aim of the paper is to introduce the `macroArray` SAS package. The package facilitates a solution that makes creation and work with macro arrays much easier. It also provides a "DATA-step-arrays-like" interface that allows use of macro arrays without complications that arise from indirect referencing. Also, the concept of a macro dictionary is presented, and all concepts are demonstrated through use cases and examples.

INTRODUCTION

A macro variable array (also MVA or just a macro array) is a jargon term for a list of macro variables with a common prefix and numerical sequential suffixes. Usually such terms have a form similar to `&&prefix&suffix` (e.g., `&&var&i`).

Every year for the last 25 years (i.e., 2000 - 2024), there has been at least one paper related to the subject of macro variable arrays published in at least one major SAS conference proceedings (this paper included). For example: [**Widawski 2000**], [**First 2001**], [**Widawski 2002**], [**Carpenter & Smith 2003**], [**Fehd 2003**], [**Clay 2004**], [**Carpenter 2005**], [**Clay 2006**], [**Lavery 2007**], [**Philp 2008**], [**Rosson 2009**], [**Russell & Tyndall 2010**], [**Gilsen 2011**], [**Li 2012**], [**Zender 2013**], [**Werner 2014**], [**Nayak 2015**], [**Wong & Short 2016**], [**Carpenter 2017**], [**Renauldo 2018**], [**Horstman 2019**], [**Huang 2020**], [**Wang 2021**], [**Walker 2022**], and [**Roudneva 2023**]. With a quick glimpse at `www.lexjansen.com`, it can be seen that articles pertaining to macro variable arrays spread they range from: simple examples, by detailed explanations and tutorials, to use cases, and eventually to development of tools to make work with MVAs easier and more comfortable.

On the one hand, MVAs (or just macro arrays) are valued by advanced SAS programmers and are often used as "driving" lists allowing to provide sequential metadata for complex or iterative programs. On the other, use of macro arrays requires advance macro programming techniques based on indirect reference, aka work with multiple ampersands (`&&`), which may be intimidating for less experienced programmers.

The aim of the article is to introduce `macroArray` SAS package. The package facilitates set of SAS macros which make creation and work with macro arrays much easier. A "data-step-arrays-like" interface, which allows use of macro arrays without complications which arise from indirect referencing,

is provided. Also an idea of macro dictionary is presented. So, according to the classification mentioned above, this article falls into the ”development of tools to make work with MVAs easier and more comfortable” category.

MACRO VARIABLE ARRAYS

This section is a ”brief” introduction to the idea of macro variable arrays. Let’s start with a definition.

As mentioned above, a *macro variable array* (an MVA or a macro array) is a list of macro variables with a common prefix and numerical suffixes. There are many ways to create a macro array. The following are some of the ”most popular” ways for creating a macro array (for color convention check Appendix A - code coloring guide).

Direct creation of an MVA can be simple assignment values to macro variables:

code: simple MVA

```
%let a1 = Macro;
%let a2 = Variables;
%let a3 = Array;
```

In this case, because of the prefix (”A”), the macro array would be referred to as ”three element macro array `a`”.

Another approach to create a macro array is to use data step and `call symputx()` routine:

code: data step approach

```
data _null_;
  length val $ 9;
  do val = "Next", "Macro", "Variables", "Array";
    i+1;
    call symputx(cats("b", i), val);
  end;
run;
```

This time we are creating ”four element macro array `b`”.

Third approach is to use SAS Proc SQL’s `into` clause to create a macro array:

code: proc sql approach

```
data have;
  input val $9.;
cards;
Proc
SQL
Macro
Variables
Array
;
run;

proc SQL;
  select val
  into :c1-
  from have
```

```
16   ;
17 quit;
```

The Proc SQL creates "five element macro array `c`".

Now when we execute:

code: preview

```
1 %put _user_;
```

we will see in the log that a dozen of macro variables were created:

the log

```
1  GLOBAL A1 Macro
2  GLOBAL A2 Variables
3  GLOBAL A3 Array
4  GLOBAL B1 Next
5  GLOBAL B2 Macro
6  GLOBAL B3 Variables
7  GLOBAL B4 Array
8  GLOBAL C1 Proc
9  GLOBAL C2 SQL
10 GLOBAL C3 Macro
11 GLOBAL C4 Variables
12 GLOBAL C5 Array
13 ...
```

The "..." indicates that there can be other macro variables created in the SAS session. In fact, since we are using Proc SQL, there are a number of macro variables automatically created (not shown).

In a nut shell this is how MVAs can be created. The following articles provides good additional learning: [**Fehd 2003**], [**Carpenter 2017**], [**Renauldo 2018**], [**Horstman 2019**], and of course the [**Carpenter 2016**] book.

When an MVA is created to retrieve values from macro variables a technique called indirect referencing is used. The indirect referencing uses the following "multiple-ampersands" approach:

`&&prefix&suffix`

The method works as follows. Let's assume we have our `a1`, `a2`, and `a3` MVA. To print all three values in the log the following macro loop does the job:

code: macro loop

```
1 %do i = 1 %to 3;
2   %put &&a&i;
3 %end;
```

(of course the loop is executed inside some macro!). In this case the prefix part is `a` and the suffix is `i`. The code execution explanation is described below:

1. Value of `i` equals `1` which is less or equal to `3`, loop starts the first iteration:
   - SAS starts scanning the `%put` statement text and processes it;
     - Two consecutive ampersands "`&&`" in the "`&&a&i`" code are recognized as a macro language triggers and are replaced with single "`&`", text kept on stack is "`&`";
     - The text "`a`" in the remaining "`a&i`" code, since it is not a macro language element, is added to the stack, which is now "`&a`";

    - The "&i" in the remaining "&i" code, is recognized as a macro variable i and its value is resolved, text "1" is added to the stack, which is now "&a1";
    - SAS gets to the end of the %put statement (the semicolon);
    - SAS "sees" that text on the stack ("&a1") still has macrotriggers (ampersands) in it and starts processing again.
  - The "&a1" in the "&a1" code, is recognized as a macro variable a1 and its value is resolved, text "Macro" is added to the stack, stack is now "Macro";
  - SAS "sees" that there are no macrotriggers in the stack text and runs the %put statement;
  - Value "Macro" is printed in the log.
  - Value of i is increased by 1;
2. Value of i equals 2 which is less or equal to 3, loop starts the second iteration:
  - SAS starts scanning the %put statement text and processes it;
    - Two consecutive ampersands "&&" in the "&&a&i" code are recognized as a macro language triggers and are replaced with single "&", text kept on stack is "&";
    - The text "a" in the remaining "a&i" code, since it is not a macro language element, is added to the stack, which is now "&a";
    - The "&i" in the remaining "&i" code, is recognized as a macro variable i and its value is resolved, text "2" is added to the stack, which is now "&a2";
    - SAS gets to the end of the %put statement (the semicolon);
    - SAS "sees" that text on the stack ("&a2") still has macrotriggers (ampersands) in it and starts processing again.
  - The "&a2" in the "&a2" code, is recognized as a macro variable a2 and its value is resolved, text "Variables" is added to the stack, stack is now "Variables";
  - SAS "sees" that there are no macrotriggers in the stack text and runs the %put statement;
  - Value "Variables" is printed in the log.
  - Value of i is increased by 1;
3. Value of i equals 3 which is less or equal to 3, loop starts the third iteration:
  - SAS starts scanning the %put statement text and processes it;
    - Two consecutive ampersands "&&" in the "&&a&i" code are recognized as a macro language triggers and are replaced with single "&", text kept on stack is "&";
    - The text "a" in the remaining "a&i" code, since it is not a macro language element, is added to the stack, which is now "&a";
    - The "&i" in the remaining "&i" code, is recognized as a macro variable i and its value is resolved, text "3" is added to the stack, which is now "&a3";
    - SAS gets to the end of the %put statement (the semicolon);
    - SAS "sees" that text on the stack ("&a3") still has macrotriggers (ampersands) in it and starts processing again.
  - The "&a3" in the "&a3" code, is recognized as a macro variable a3 and its value is resolved, text "Array" is added to the stack, stack is now "Array";
  - SAS "sees" that there are no macrotriggers in the stack text and runs the %put statement;
  - Value "Array" is printed in the log.
  - Value of i is increased by 1;
4. Value of i equals 4 which is not less or equal to 3, loop stops. The end.

The result in the log is:

the log

```
1 Macro
2 Variables
3 Array
```

If you are "junior" SAS programmer and feel a bit overwhelmed by above explanation - do not be. All that text can be boiled down to the following four rules:

- double ampersands (`&&`) are replaced with a single one (`&`),
- macro variables (`&macVar`) are resolved to the value (`value`),
- any non-macro text is kept "as is",
- repeat until there are no more macrotriggers.

When it goes to creation of MVAs, it can be seen that each of the above three methods has some "shortcomings". In the first one, it is obvious that the shortcoming is that it was created "by-hand". The second requires jumping back from the macro language level to the 4GL data step. The third in addition to "4GL back jumping" also requires a data set with input values. Further more, for inexperienced users the use of "multiple-ampersands" approach to resolve macro variables data may be a bit "intimidating" or even obscured[1] and for experienced one it may cause "my-code-is-a-mess" thinking. For all that "little pains" the SAS users community has solutions!

STANDING ON THE SHOULDERS OF GIANTS

The SAS community is a very active one, so in no time, solutions allowing to "improve" convenience of work started to appear. Those of the highest value were articles presented by Ted Clay ([**Clay 2004**] and [**Clay 2006**]) where the `%array()` and the `%do_over()` macros were presented. The purpose of the `%array()` macro is to create MVAs in an easy and convenient way and the purpose of the `%do_over()` macro is to allow practical looping over MVAs created by the `%array()`. For example:

code: Ted Clay's approach

```
data Have;
  input dayName :$12. value;
cards;
Monday 1
Tuesday 2
Wednesday 3
Thursday 4
Friday 5
Saturday 6
Sunday 7
;
run;

%ARRAY(days, VALUES=Monday Wednesday Friday Sunday)

data Want;
  set Have end=end;
    where dayName in ( %DO_OVER(days, PHRASE="?"));
  total + value;
  if end then output;
run;
```

The `%array()` creates four global macro variables `days1=Monday`, `days2=Wednesday`, `days3=Friday`, `days4=Sunday`, and additional (technical) variable `daysN` with value 4, which keeps the information

[1] Despite the "awkwardness" and "this-looks-to-hard-to-me" the author highly recommend putting effort and learning/understanding in depth how the process works.

about the MVA length. The `%do_over()`, with help of `daysN`, executes internal loop over `days*` and produce a text string:

```
"Monday" "Wednesday" "Friday" "Sunday"
```

which is used inside the `where` clause. Further more, the `%array()` macro also allows for the creation of MVAs from data sets.

The section title "Standing on the shoulders of giants" (a metaphor which means "using the understanding gained and work done by major thinkers working in the field before you in order to make progress") is fully justified here since the first contact with those macros, used to solve problems on the SAS-L discussion list and, in consequence, lecture of Ted Clay's articles gave birth to the following thoughts in the authors head:

1. "This is brilliant!"
2. "How does it work?!" and (after answering the question)
3. "Can its functionality be improved?"

Eventually, the development of the `macroArray` package was the result. One more remark, it is "giants", not "giant", because the source code clearly states: "*Authors:* **Ted Clay, David Katz**"!

So, how was it with the `macroArray` package development in the first place? After reading through Clay's and Katz's `%array()` code the first observation was that the code is not "pure macro code" [2] (to be more precise, the part responsible for producing MVAs from data sets). According to the code information notes, the last modification of the Clay's and Katz's `%array()` macro was in August 2006. At that point the famous `DoSubL()` function (see [**Langston 2013**], [**McMullen 2020**], or [**Jablonski(1) 2023**]) was not at their disposal to utilize the "macro-function sandwich" technique. Fortunately, when I was taking my first attempts to write my own version in January 2019 (see SAS-L post: "%array() macro and dosubl() function" - January 18, 2019, although do *not* use version of the code from that post, it is quite obsolete now) I was fortunate to have the `DoSubL()` at hand. My experience with the `DoSubL()` led me to my first, and then several consecutive, question.

So the first question was: can we have `%array()` working as "pure macro code" (even for creating macro arrays from data sets)? The answer is yes.

The next observation was, macro arrays are "one based", i.e., the first element has always index 1. The second question was: can an MVA created by `%array()` start indexing from an arbitrary non-negative integer (e.g., 0, 2, 3, etc.)? Here, the answer is also yes.

Following idea was: we can create multiple MVAs from multiple variables from a data set. But can we also create MVAs with lists of unique values? Yes.

When macro array is created its values still have to be called by `&&prefix&suffix` approach. Can we make this call to the macro array value more "convenient", e.g., similar to the data step array, like: (`prefix[suffix]`)? Yes.

Can we use SAS functions to alter, modify, or generate MVA values? Yes.

When creating an MVA, can we have syntax similar to the one known form 4GL? Yes we can.

And one more, very important question. Can all those "yes we can" extension be added keeping backward compatibility? Also, yes.

[2] The "pure macro code" means it is a code which executes 100% on the macro processor level and does not produce any `data` or `proc` step code for the compilation.

So, it is nice that all the answers were: yes we can! But now, our little internal "Bob the Builder" wants details. Explanation to the questions stated above can be found in the next section where the `macroArray` package is described.

THE `macroArray` PACKAGE

In this section we are going to discuss the `macroArray` package, its components, capabilities, and possible use cases.

The latest (and historical) version of the `macroArray` package, with its documentation, parameters description and examples, can be found at:

https://github.com/SASPAC/macroarray

[**Note**:] To start playing with any SAS Package, not only the `macroArray` package, the package has to be first installed and loaded into the SAS session. The Appendix B - install the SAS Packages Framework and the `macroArray` package provides detailed instruction of the process. Detailed information about SAS Packages idea can be found in [**Jablonski 2020**], [**Jablonski 2021**], and [**Jablonski(2) 2023**].

Assuming we are ready to go, let's see what the `macroArray` package has to offer.

At the moment the package contains 14 macros, which are: `%appendArray()`, `%appendCell()`, `%array()`, `%concatArrays()`, `%deleteMacArray()`, `%do_over()`, `%do_over2()`, `%do_over3()`, `%make_do_over()`, `%mcHashTable()`, `%mcDictionary()`, `%QzipArrays()`, `%zipArrays()`, and `%sortMacroArray()`.

Rewriting `macroArray` package's documentation would be rather silly idea, especially since the documentation is publicly available (see the github link above). Instead, we discover the `macroArray` package by "getting our hands dirty", i.e., by going through a bunch of examples and use cases presenting package's features and capabilities.

The `%array()` macro.

The first macro we discuss is the `%array()`[3] macro. It allows us to create MVAs, either "from code" or from a data set.

Let's start with something simple. We want to create six element MVA "named" `X` with values from 101 to 106. For now, we will use the `%put _user_;` statement to print created macro variables.

code: %array() example No. 1

```
%array(X[6] (101:106))

%put _user_;
```

In the log we can see the following:

the log

```
1    %array(X[6] (101:106))
NOTE:[ARRAY] 6 macrovariables created
2
3    %put _user_;
GLOBAL SYSLOADEDPACKAGES macroArray(X.Y.Z)
GLOBAL X1 101
GLOBAL X2 102
```

[3] To be clear, although the macros `%array()` and `%do_over()` were designed to replicate the functionality of Clay and Katz's macros, their source code was written from scratch. Only the names are the same.

```
GLOBAL X3 103
GLOBAL X4 104
GLOBAL X5 105
GLOBAL X6 106
GLOBAL XHBOUND 6
GLOBAL XLBOUND 1
GLOBAL XN 6
```

Let's take a look at what we see.
First, there is a note issued by `%array()` macro saying that six macro variables were created.
Second, syntax of the code looks "4GL-ish". Indeed when we like to create a data step array we write almost exactly the same:

code: 4GL array

```
data _null_;
  array X[6] (101:106);
run;
```

This way of writing code is possible thanks to the `DoSubL()` function working under the hood and giving us a bit of "syntactic sugar" flavor which allows us to learn how to use the `%array()` macro faster.
Third, is a printout of user's macro variables. We can see six global macro variables `X1` to `X6` with values from 101 to 106 which "constitute" the macro array. Additionally, we can see macro variable `XN` which indicates the number of macro array elements, similarly to the original Clay's and Katz's `%array()` macro. What is new are two macro variables `XLBOUND` and `XHBOUND` which indicates lower and, respectively, higher index of the macro array, in this example they are 1 and 6 respectively. We discuss `XLBOUND` and `XHBOUND` more in a bit, but now let's take a look at one more macro variable, namely the `SYSLoadedPackages`.

We are using the `macroArray` package which was loaded into the SAS session by the SAS Packages Framework. The `SYSLoadedPackages` is special technical macro variable generated/updated by the framework when a package is loaded into the SAS Session. The `SYSLoadedPackages` keeps information about all SAS packages loaded in current SAS session. The value of the `SYSLoadedPackages` has the following form: "`package1(version) package2(version) ... packageN(version)`". In the future log printouts of user's macro variables, for brevity, that one will be skipped.

[**NOTE:** About array name length] The naming convention for macro arrays is "SAS standard" with the respect to the following restrictions: cannot be empty, *cannot* be longer than "$32 - max(6,$ *the number of digits in the biggest suffix value*$)$". In other words, if the maximum suffix is 123456789 then maximum length of a macro array name can be up to 23 ($32 - max(6,9)$) symbols[4]. The "6" is from the "HBOUND" and "LBOUND" suffixes. There is one more "special case", when macro array's name is single underscore, but this case is explained separately and in details in the documentation.

Before further discussion, let's take a look at the next example. We create the `days` macro array, with values: `Monday`, `Wednesday`, `Friday`, and `Sunday`, like in the example from previous section. Since we can do it on at least two ways, the first MVA will be named: `days_A` and the second: `days_B`.

code: %array() example No. 2

```
%array(days_A[*] Monday Tuesday Wednesday Saturday, vnames=Y)

%array(days_B[4] $ 20 ("Monday" "Tuesday" "Wednesday" "Saturday"))
```

[4] By "symbols" we mean the 26 letters of the Roman alphabet, ten digits from 0 to 9 and the underscore character.

```
4
5  %put _user_;
```

The log shows the following:

the log - with "vnames=" parameter

```
1    %array(days_A[*] Monday Tuesday Wednesday Saturday, vnames=Y)
NOTE:[ARRAY] 4 macrovariables created
2
3    %array(days_B[4] $ 20 ("Monday" "Tuesday" "Wednesday" "Saturday"))
NOTE:[ARRAY] 4 macrovariables created
4
5    %put _user_;
GLOBAL DAYS_A1 Monday
GLOBAL DAYS_A2 Tuesday
GLOBAL DAYS_A3 Wednesday
GLOBAL DAYS_A4 Saturday
GLOBAL DAYS_AHBOUND 4
GLOBAL DAYS_ALBOUND 1
GLOBAL DAYS_AN 4
GLOBAL DAYS_B1 Monday
GLOBAL DAYS_B2 Tuesday
GLOBAL DAYS_B3 Wednesday
GLOBAL DAYS_B4 Saturday
GLOBAL DAYS_BHBOUND 4
GLOBAL DAYS_BLBOUND 1
GLOBAL DAYS_BN 4
```

We can see in both cases that a macro array was generated, but in each case a different approach was used.

In the first approach we use the `vnames=Y` parameter and the fact that values: `Monday`, `Wednesday`, `Friday`, and `Sunday` can be used as data step variables names. The `vnames=Y` parameter instructs the `%array()` macro to take variable's names as MVA's values instead "real" values stored in variables (the default behavior). If we do not use the `vnames=` parameter or set its value to anything other then `Y`, the result is:

the log - no "vnames=" parameter

```
1    %array(days_A[*] Monday Tuesday Wednesday Saturday)
NOTE:[ARRAY] 4 macrovariables created
2
3    %put _user_
GLOBAL DAYS_A1 .
GLOBAL DAYS_A2 .
GLOBAL DAYS_A3 .
GLOBAL DAYS_A4 .
GLOBAL DAYS_AHBOUND 4
GLOBAL DAYS_ALBOUND 1
GLOBAL DAYS_AN 4
```

The reason we get "periods" is because (according to 4GL rules) the array under the hood is considered to be numeric. Bottom line is, if values (e.g., week days names) you want to use for MVA values "satisfy" SAS variables naming convention you can do it with `vnames=Y`.

In the second approach by writing "`[4] $ 20`" we are declaring a character array of size four with initial values which are then used to generate the MVA. This approach does not have "limitations" of the first one. Furthermore since values are provided in quotes it can be very handy way of "masking" special characters like ampersand, percent, semicolon or comma in macro array values, for example:

the log - "specials"

```
1    %array(days_B[4] $ 20 ('&Monday.' '%Tuesday()' "Wed;nes;day;" "Sa,tur,day")
NOTE:[ARRAY] 4 macrovariables created
2
3    %put _user_
GLOBAL DAYS_B1 &Monday.
GLOBAL DAYS_B2 %Tuesday()
GLOBAL DAYS_B3 Wed;nes;day;
GLOBAL DAYS_B4 Sa,tur,day
GLOBAL DAYS_BHBOUND 4
GLOBAL DAYS_BLBOUND 1
GLOBAL DAYS_BN 4
```

Obviously the `"Wed;nes;day;"` text can be replaced with something like: `"proc print data=sashelp.class; var name age; run;"`.

In the examples above values of the MVA were a "static text", but this does not have to be the case. The `%array()` macro can be extended with the use of SAS functions that can be used to generate macro array values. A dedicated parameter, `function=`, can be used to assign the array's values. The `function=` parameter expects a SAS 4GL function, composition of functions, or an expression using functions. If we would like to use array's index in an expression the index value is kept in the `_i_` variable[5]. *Important thing to remember* is that if the type of an array it numeric, then the value returned by the expression in `function=` parameter must "align" (be numeric too). Let's look at log output for some examples:

the log - with "function=" parameter

```
1    /* three constant expression, equiv. of %array(e[1:3] $ (3*"A") ) */
2    %array(e[1:3] $, function = "A" )
NOTE:[ARRAY] 3 macrovariables created
3
4    /* first six integer powers of 2 */
5    %array(f[0:5], function = (2**_i_) )
NOTE:[ARRAY] 6 macrovariables created
6
7    /* five random numbers form uniform distribution on (0,1) interval */
8    %array(g[0:4], function = ranuni(123) )
NOTE:[ARRAY] 5 macrovariables created
9
10   /* a formated list of twelve months */
11   %array(h[0:11] $ 11, function = put(intnx("MONTH", '19may2024'd, _i_), yymmd.))
NOTE:[ARRAY] 12 macrovariables created
12
13   /* Fibonacci sequence, first ten elements */
14   %array(i[10] (10*0)
15           ,function = ifn(_i_<2, 1, sum(i[max(_i_-2,1)], i[max(_i_-1,2)])))
NOTE:[ARRAY] 10 macrovariables created
```

[5]The `_i_` variable is used internaly inside the `%array()` macro processing.

```
16
17   %put _user_;
GLOBAL E1 A
GLOBAL E2 A
GLOBAL E3 A
GLOBAL EHBOUND 3
GLOBAL ELBOUND 1
GLOBAL EN 3

GLOBAL F0 1
GLOBAL F1 2
GLOBAL F2 4
GLOBAL F3 8
GLOBAL F4 16
GLOBAL F5 32
GLOBAL FHBOUND 5
GLOBAL FLBOUND 0
GLOBAL FN 6

GLOBAL G0 0.7503960881
GLOBAL G1 0.3209120251
GLOBAL G2 0.178389649
GLOBAL G3 0.9060333813
GLOBAL G4 0.3571170775
GLOBAL GHBOUND 4
GLOBAL GLBOUND 0
GLOBAL GN 5

GLOBAL H0 2024-05
GLOBAL H1 2024-06
GLOBAL H2 2024-07
GLOBAL H3 2024-08
GLOBAL H4 2024-09
GLOBAL H5 2024-10
GLOBAL H6 2024-11
GLOBAL H7 2024-12
GLOBAL H8 2025-01
GLOBAL H9 2025-02
GLOBAL H10 2025-03
GLOBAL H11 2025-04
GLOBAL HHBOUND 11
GLOBAL HLBOUND 0
GLOBAL HN 12

GLOBAL I1 1
GLOBAL I2 1
GLOBAL I3 2
GLOBAL I4 3
GLOBAL I5 5
```

```
GLOBAL I6 8
GLOBAL I7 13
GLOBAL I8 21
GLOBAL I9 34
GLOBAL I10 55
GLOBAL IHBOUND 10
GLOBAL ILBOUND 1
GLOBAL IN 10
```

The `function=` parameter with all its "functionality" is not all that is. There are two more (optional) counterpart parameters: `before=` and `after=`. The `before=` expects a function or an expression to be added before looping through the array. The `after=` expects a function or an expression to be added after looping through the array. If we want to provide a series of statements to `before=` or `after=` they should be separated by semicolons. The following example generates macro array `j` with sorted random numbers in range from zero to one, generated by the `rand()` function from uniform distribution. The seed for the `rand()` function is set with the help of `before=` parameter. Sorting, change of the first and the last value, and a "debugging printout" are done with `after=` parameter:

code: %array() example No. 3

```
%array(j[10]
  ,function = round(rand('Uniform',0,1),0.001)
  ,before = call streaminit(42)
  ,after = call sortn(of j[*]); j[1]=0; j[10]=1; put _all_
)

%put _user_
```

The log presents the following content:

the log - with "before=" and "after=" parameters

```
1    %array(j[10]
2      ,function = round(rand('Uniform',0,1),0.001)
3      ,before = call streaminit(42)
4      ,after = call sortn(of j[*]); j[1]=0; j[10]=1; put _all_
5    )

j1=0 j2=0.137 j3=0.219 j4=0.513 j5=0.66 j6=0.728 j7=0.747 j8=0.753 j9=0.901 j10=1
_I_=11 _ERROR_=0 _N_=1
NOTE:[ARRAY] 10 macrovariables created
6
7    %put _user_;
GLOBAL J1 0
GLOBAL J2 0.137
GLOBAL J3 0.219
GLOBAL J4 0.513
GLOBAL J5 0.66
GLOBAL J6 0.728
GLOBAL J7 0.747
GLOBAL J8 0.753
GLOBAL J9 0.901
GLOBAL J10 1
GLOBAL JHBOUND 10
```

```
GLOBAL JLBOUND 1
GLOBAL JN 10
```

So at this point an MVA created by `macroArray` package has all properties provided by the Clay's and Katz's `%array()` macro, so we can also use it with their version of `%do_over()`.

Next extension of the `%array()` macro is the possibility to index "coded" macro arrays not only starting from one but from any non-negative integer (e.g., 0, 2, 3, etc.), for example:

code: %array() example No. 4

```
%array(k[5:7] $ ("fifth" "sixth" "seventh"))

%put _user_;
```

In the log we see:

the log - with "arbitrary index"

```
1    %array(k[5:7] $ ("fifth" "sixth" "seventh"))
NOTE:[ARRAY] 3 macrovariables created
2
3    %put _user_;
GLOBAL K5 fifth
GLOBAL K6 sixth
GLOBAL K7 seventh
GLOBAL KHBOUND 7
GLOBAL KLBOUND 5
GLOBAL KN 3
```

In case we decide to start indexing from a negative integer (e.g. -3) then the index range value in the variable `_i_` is used "as is", i.e., starts from provided negative value, but... and *this has to be highlighted(!)*, the index range for macro array is *automatically shifted to start at zero*! For example if we want to get powers of two with exponents from -3 to 3 in macro array we can do the following:

code: %array() example No. 5

```
%array(l[-3:3], function = 2**_i_)

%put _user_;
```

In this case the log shows:

the log - with "negative index"

```
1    %array(l[-3:3], function = 2**_i_)
NOTE:[ARRAY] 7 macrovariables created
2
3    %put _user_;
GLOBAL L0 0.125
GLOBAL L1 0.25
GLOBAL L2 0.5
GLOBAL L3 1
GLOBAL L4 2
GLOBAL L5 4
GLOBAL L6 8
GLOBAL LHBOUND 6
```

```
GLOBAL LLBOUND 0
GLOBAL LN 7
```

So the values assigned are 2^{-3}, 2^{-2}, ..., 2^2, 2^3 but the suffixes goes from zero to six.
The reason for implementing such behavior is based on the following thoughts. The only characters allowed for macro variables names are letters, digits and underscore("_"). What symbol could be used to indicate minus sign in negative index? It cannot be a number or a letter, the first are already reserved for suffix part the second for prefix. The only "natural candidate" which pops up is the underscore. But if we decide to use underscore as an indicator of negative value, then from looking at the following macro array `M_1=2`, `M_2=1` you cannot tell which of those two examples: `%array(m[-2:-1] (1 2))` and `%array(m_[1:2] (2 1))`, was used to produce the MVA. In other words, we cannot tell if it is array `M_` with index `1`, or if it is array `M` with index `-1`.

But macro arrays created from data sets are indexed from one! Right! Macro variable arrays can be created from a data set, we didn't mention this feature yet.

The `%array()` macro allows us to create a macro array from a variable in a data set. Even better, multiple macro arrays can be created based on different variables from that data set. Additionally, it is possible to request a macro array with only distinct values extracted from a variable. To create MVA from a variable in a data set two parameters should be used. The first one, `ds=`, points to a data set, the second, `vars=`, points to variables. The `ds=` parameter takes precedence over other parameters, i.e., if both `ds=` is not empty (points to a data set) and "by code"[6] array definition is provided then the `ds=` dedicated code is executed. Data set in the `ds=` can be provided with data set options in parentheses but the data set can be (like in the "Highlander" movie) only one. On the other hand, the `vars=` parameter allows for multiple values, this will be discussed next, but now let's take a look at the following example:

code: %array() example No. 6

```
%array(ds=sashelp.class(where=(age=13)), vars=name)

%put _user_;
```

The condition in the `where=` data set option selects three observations from `sashelp.class` data set and values of the `name` variable are taken for MVA values, the name of the variable is used for MVA name. Log shows the following result:

the log - with "ds=" and "vars=" parameter

```
1    %array(ds=sashelp.class(where=(age=13)), vars=name)
NOTE:[ARRAY] 3 macrovariables created
2
3    %put _user_;
GLOBAL NAME1 Alice
GLOBAL NAME2 Barbara
GLOBAL NAME3 Jeffrey
GLOBAL NAMEHBOUND 3
GLOBAL NAMELBOUND 1
GLOBAL NAMEN 3
```

If we decide to create multiple MVAs we simply list variables we want to use separated by a space, for example like this:

[6] A "by code" array is an array declared within the macro call

code: %array() example No. 7

```
%array(ds=sashelp.class(where=(age=13)), vars=name height weight)

%put _user_;
```

Log from execution is the following:

the log - ith multiple variables

```
1    %array(ds=sashelp.class(where=(age=13)), vars=name height weight)
NOTE:[ARRAY] 9 macrovariables created
2
3    %put _user_;
GLOBAL HEIGHT1 56.5
GLOBAL HEIGHT2 65.3
GLOBAL HEIGHT3 62.5
GLOBAL HEIGHTHBOUND 3
GLOBAL HEIGHTLBOUND 1
GLOBAL HEIGHTN 3

GLOBAL NAME1 Alice
GLOBAL NAME2 Barbara
GLOBAL NAME3 Jeffrey
GLOBAL NAMEHBOUND 3
GLOBAL NAMELBOUND 1
GLOBAL NAMEN 3

GLOBAL WEIGHT1 84
GLOBAL WEIGHT2 98
GLOBAL WEIGHT3 84
GLOBAL WEIGHTHBOUND 3
GLOBAL WEIGHTLBOUND 1
GLOBAL WEIGHTN 3
```

First we observe three brand new MVAs created, the second observation takes us to the log note printed by the macro. The note reports *total* number of macro variables created.

The `vars=` parameter has two more features. The first, is a possibility to select only unique values from a variable. It can be done by adding a vertical bar character ("|") *directly after* the variable name in the list, in other words "no space in-between", like in this example:

code: %array() example No. 8

```
%array(ds=sashelp.class, vars=age|)

%put _user_;
```

Log shows the following:

the log - with unique values

```
1    %array(ds=sashelp.class, vars=age|)
NOTE:[ARRAY] 6 macrovariables created
2
3    %put _user_;
GLOBAL AGE1 15
```

```
GLOBAL AGE2 12
GLOBAL AGE3 16
GLOBAL AGE4 13
GLOBAL AGE5 14
GLOBAL AGE6 11
GLOBAL AGEHBOUND 6
GLOBAL AGELBOUND 1
GLOBAL AGEN 6
```

We can see six unique values selected. Order of returned values, as can be seen, is random. In general: if there is "|" inserted behind a variable name unique values are selected, and if there is nothing or *the pound sign* ("#") behind a variable name then values from all observations are selected.

A curious reader instantly asks: Why additional symbol (the pound sign)? The vertical bar(|) and "nothing" perfectly good distinguish "select distinct vs. select all" values. Well, it would be enough but *only* in situations when we want MVA name to be the same as the data set variable's name. And this is the second feature the `vars=` parameter has. We can use values from a variable but we do not need to use its name, we can change MVA's name by setting an alias. The syntax is:

```
variable1<| or #<alias1>> <variable2<| or #<alias2>>> ...  <variableN<| or #<aliasN>>>
```

The `<...>` indicates optional element. Basically, the `variableA|aliasA` means: select distinct values from `variableA` into MVA named `aliasA`, and the `variableB#aliasB` means: select all values from `variableB` into MVA named `aliasB`. The best is to look at the next example in which we use `sashelp.class` and create macro array `N` with *all* values from variable `name` and macro array `A` with *unique* values from variable `age`:

code: %array() example No. 9

```
%array(ds=sashelp.class, vars=name#N age|A)

%put _user_;
```

And, as we would expect, in the log we see 25 variables created:

the log - with "vars=" parameter in full

```
1    %array(ds=sashelp.class, vars=name#N age|A)
NOTE:[ARRAY] 25 macrovariables created
2
3    %put _user_;
GLOBAL A1 15
GLOBAL A2 12
GLOBAL A3 16
GLOBAL A4 13
GLOBAL A5 14
GLOBAL A6 11
GLOBAL AHBOUND 6
GLOBAL ALBOUND 1
GLOBAL AN 6
GLOBAL N1 Alfred
GLOBAL N2 Alice
GLOBAL N3 Barbara
...
GLOBAL N17 Ronald
GLOBAL N18 Thomas
```

```
GLOBAL N19 William
GLOBAL NHBOUND 19
GLOBAL NLBOUND 1
GLOBAL NN 19
```

We all know that in the macro language world every value is a text so no quotes are needed. But even despite that from time to time there is a need that value must be presented in quotes (double or single). That is why one more parameter, q=, which indicate if quotes should be added, is available. The parameter can take two values, 2 to indicate quoting with double quotes and 1 to indicate quoting with apostrophes (single quotes). Parameter works both with "coded" macro arrays and with "from data set" arrays. Log for two simple examples looks like this:

the log - with "q=" parameter

```
1    %array(Letters[5] $ 3, function = byte(rank('A')+_I_-1) , q=2)
NOTE:[ARRAY] 5 macrovariables created
2
3    %array(ds=sashelp.class, vars=name, q=1)
NOTE:[ARRAY] 19 macrovariables created
4
5    %put _user_;
GLOBAL LETTERS1 "A"
GLOBAL LETTERS2 "B"
GLOBAL LETTERS3 "C"
GLOBAL LETTERS4 "D"
GLOBAL LETTERS5 "E"
GLOBAL LETTERSHBOUND 5
GLOBAL LETTERSLBOUND 1
GLOBAL LETTERSN 5
GLOBAL NAME1 'Alfred'
GLOBAL NAME2 'Alice'
GLOBAL NAME3 'Barbara'
...
GLOBAL NAME17 'Ronald'
GLOBAL NAME18 'Thomas'
GLOBAL NAME19 'William'
GLOBAL NAMEHBOUND 19
GLOBAL NAMELBOUND 1
GLOBAL NAMEN 19
```

[**Note**:] If you are working with BIG macro arrays do not forget to verify your session setting for macro memory limits. Run:

code: session setting for macro memory

```
proc options group = macro;
run;
```

to verify the following options:

- MEXECSIZE - specifies the maximum macro size that can be executed in memory.
- MSYMTABMAX - specifies the maximum amount of memory available to the macro variable symbol table or tables.
- MVARSIZE - specifies the maximum size for a macro variable that is stored in memory.

In the final part of this subsection we discuss one more feature, unique to the macroArray package's implementation of the %array() macro. Up to now, to display macro array's elements in the log we use the %put _user_; trick. But what to do when we want to use MVA's elements separately, e.g., in a loop of some sort? As we already know, the "structure" of variables in a macro array has the form: [prefix][suffix] and when we want to call their values we have to use indirect reference. To call elements of the macro array ARR in the example below we use the &&prefix&suffix construction in the %do-loop:

code: %array() example No. 10

```
%macro someMacro();
  %array(ARR[1:3] $ ("first" "second" "third"))

  %do i=1 %to 3;
    %put for &=i. value is &&ARR&i ;
  %end;
%mend someMacro;

%someMacro()
```

In the log, printout from the %someMacro() execution has the following form:

the log - getting values

```
1    %someMacro()
NOTE:[ARRAY] 3 macrovariables created
for I=1 value is first
for I=2 value is second
for I=3 value is third
```

To make macro arrays behavior even more "4GL arrays" alike the macarray=<Y,N> parameter was introduced. The purpose of the macarray=, when set to Y (the default is N), is to generate a *new macro* named the same as the MVA name. The new macro that is created allows us to call to macro arrays elements in a similar fashion we call array elements in the data step, with the difference being that now the percent symbol (%) precedes the reference. The example above can be modified the following way:

code: %array() example No. 11

```
%macro someMacro2();
  %array(ARR[1:3] $ ("first" "second" "third"), macarray=Y)

  %do i=1 %to 3;
    %put for &=i. value is %ARR(&i) ;
  %end;
%mend someMacro2;

%someMacro2()
```

And in the log, printout from the %someMacro2() execution looks exactly the same:

the log - getting values with "macarray=" parameter

```
1    %someMacro2()
NOTE:[ARRAY] 3 macrovariables created
for I=1 value is first
for I=2 value is second
for I=3 value is third
```

Clearly the `%ARR(&i)` is more "array like" looking code than the `&&ARR&i` is, isn't it? Plus the added benefit is you don't have to remember how many `&` are needed.

The parameter works with data set generated macro arrays too. And when used with lower and high bound variables it works even better, like in this example:

code: %array() example No. 12

```
%macro someMacro3();
  %array(ds=sashelp.class, vars=age|ARR, macarray=Y)

  %do i=&ARRLBOUND. %to &ARRHBOUND.;
    %put for &=i. value is %ARR(&i) ;
  %end;
%mend someMacro3;
```

Log printout:

the log - with "macarray=" and "ds=" parameters

```
1    %someMacro3()
NOTE:[ARRAY] 6 macrovariables created
for I=1 value is 15
for I=2 value is 12
for I=3 value is 16
for I=4 value is 13
for I=5 value is 14
for I=6 value is 11
2    %put _user_;
GLOBAL ARR1 15
GLOBAL ARR2 12
GLOBAL ARR3 16
GLOBAL ARR4 13
GLOBAL ARR5 14
GLOBAL ARR6 11
GLOBAL ARRHBOUND 6
GLOBAL ARRLBOUND 1
GLOBAL ARRN 6
```

The "macro-array" macro created when the `macarray=` parameter is in use has one more convenient feature. It allows us to reassign values of existing macro variables. The process is very straightforward, all we need to do is to set the "macro-array" macro's second argument to "`i`" (like "insert"). Here is the example:

code: %array() example No. 13

```
%array(ABC[3] (1:3), macarray=Y);

%put _user_;
%let %ABC(2,i) = 99999;
%put _user_;
```

With log print showing:

the log - modifying macro array values

```
1    %array(ABC[3] (1:3), macarray=Y);
NOTE:[ARRAY] 3 macrovariables created
```

```
3  2
4  3    %put _user_;
5  GLOBAL ABC1 1
6  GLOBAL ABC2 2
7  GLOBAL ABC3 3
8  GLOBAL ABCHBOUND 3
9  GLOBAL ABCLBOUND 1
10 GLOBAL ABCN 3
11 4    %let %ABC(2,i) = 99999;
12 5    %put _user_;
13 GLOBAL ABC1 1
14 GLOBAL ABC2 99999
15 GLOBAL ABC3 3
16 GLOBAL ABCHBOUND 3
17 GLOBAL ABCLBOUND 1
18 GLOBAL ABCN 3
```

The `macarray=` parameter provides one more useful feature. If its value is set to `M` (”make”) then for a given array name the macro symbols table is scanned for all macro variables with a prefix like the array name and numeric suffixes. From all macro variables of that form found, the minimum and the maximum index is determined and all non-existing sequential global macro variables between the minimum and maximum index values are created. Also a macro is generated, in the same fashion as one is generated for the `macarray=Y` value. Here is log printout from a small example:

the log - with macarray=M

```
1  1    %let ARR1=1;
2  2    %let ARR3=3;
3  3    %let ARR5=5;
4  4
5  5    %put _user_;
6  GLOBAL ARR1 1
7  GLOBAL ARR3 3
8  GLOBAL ARR5 5
9  6    %array(ARR,macarray=m)
10 NOTE:[ARRAY] 5 macrovariables created
11 7    %put _user_;
12 GLOBAL ARR1 1
13 GLOBAL ARR2
14 GLOBAL ARR3 3
15 GLOBAL ARR4
16 GLOBAL ARR5 5
17 GLOBAL ARRHBOUND 5
18 GLOBAL ARRLBOUND 1
19 GLOBAL ARRN 5
```

This ends the discussion about features and capabilities of the `%array()` macro.

The `%deleteMacArray()` macro.

Purpose of the `%deleteMacArray()` macro is very simple, it is a ”house keeping” macro. When macro

array is created with help of the %array() macro the macro variables are set as global, even if the %array() macro execution creates an MVA on seventeenth, or forty-second level of macro nesting depth. This may, when less attention is given, produce overwrite errors. To minimize probability of such error the best approach is to delete macro array as soon as it is no longer needed. The "possible use case" code could be something like the following one:

code: clean after work

```
%macro someMacroX();

  /* declare someMacroArray */
  %array(<definition of someMacroArray>, macarray=Y)

  /* code using someMacroArray */
  %do i=&someMacroArrayLBOUND. %to &someMacroArrayHBOUND.;
    /* <... call to %someMacroArray(&i) ...> */
  %end;

  /* delete someMacroArray */
  %deleteMacArray(someMacroArray, macarray=Y)

%mend someMacroX;
```

The second argument of the %deleteMacArray() macro, the macarray=<Y,N>, indicates should a macro associated with macro array be deleted too. Default is N meaning "no".

If we specify a macro array to delete and there isn't one an error in the log is produced. If we indicate macarray=Y and there is no macro associated with the MVA the %deleteMacArray() macro still works and deletes the macro array specified in the first argument.

The %sortMacroArray() macro.

Another "house keeping" macro is the %sortMacroArray() macro. As the name suggests the macro allows to sort macro array's values, for example like this case:

code: sorting macro array

```
%array(n[6] $ 3 ("C33" "B22" "A11" "A01" "A02" "X42"))

%put _user_;

%sortMacroArray(n)

%put _user_;
```

Log shows the following:

the log - with "macarray=" and "ds=" parameters

```
1    %array(n[6] $ 3 ("C33" "B22" "A11" "A1" "A2" "X42"))
NOTE:[ARRAY] 6 macrovariables created
2
3    %put _user_;
GLOBAL N1 C33
GLOBAL N2 B22
GLOBAL N3 A11
GLOBAL N4 A1
```

```
GLOBAL N5 A2
GLOBAL N6 X42
GLOBAL NHBOUND 6
GLOBAL NLBOUND 1
GLOBAL NN 6
4
5    %sortMacroArray(n)
6
7    %put _user_;
GLOBAL N1 A1
GLOBAL N2 A2
GLOBAL N3 A11
GLOBAL N4 B22
GLOBAL N5 C33
GLOBAL N6 X42
GLOBAL NHBOUND 6
GLOBAL NLBOUND 1
GLOBAL NN 6
```

Though macro works as "pure macro code", internally the `Proc SORT` does the job, the sorting sequence can be modified by the `sortseq=` parameter. Its default value is `LINGUISTIC(NUMERIC_COLLATION = ON)` which sort data that contain numbers in the values as if they were numbers (e.g., "A2" before "A10").

Additionally, the `outSet=` macro parameter allows to store sorted values with the index "after sorting" in a data set pointed by the parameter. An example of sorting the MVA `n` from the previous example and saving its data in the `sortedData` data set looks like this:

code: sorting macro array and storing sorted data

```
%sortMacroArray(n,outSet=sortedData)
```

Since under the hood `Proc SORT` is used the macro has one limitation though, maximum length of sorted values is 32767 bytes.

The %appendArray() and the %concatArrays() macros.

Two more macros that "organize macro arrays world" are `%appendArray()` and `%concatArrays()`. The first macro (in fact a wrapper over the second) allows to append one macro array's values at the end of another macro array. The second macro does the same as the first one does, but also deletes the second array's elements after appending. The log illustrates this situation the best:

the log - appending one to another

```
1    %array(o[2:4] $ 1 ("A" "B" "C"))
NOTE:[ARRAY] 3 macrovariables created
2
3    %array(p[3] (1 2 3))
NOTE:[ARRAY] 3 macrovariables created
4
5    %put _user_;
GLOBAL O2 A
GLOBAL O3 B
GLOBAL O4 C
GLOBAL OHBOUND 4
```

```
GLOBAL OLBOUND 2
GLOBAL ON 3
GLOBAL P1 1
GLOBAL P2 2
GLOBAL P3 3
GLOBAL PHBOUND 3
GLOBAL PLBOUND 1
GLOBAL PN 3
6    %appendArray(o, p);
7    %put _user_;
GLOBAL O2 A
GLOBAL O3 B
GLOBAL O4 C
GLOBAL O5 1
GLOBAL O6 2
GLOBAL O7 3
GLOBAL OHBOUND 7
GLOBAL OLBOUND 2
GLOBAL ON 6
GLOBAL P1 1
GLOBAL P2 2
GLOBAL P3 3
GLOBAL PHBOUND 3
GLOBAL PLBOUND 1
GLOBAL PN 3
```

And the second one:

the log - concatenating two

```
1    %array(q[2:4] $ 1 ("E" "F" "G"))
NOTE:[ARRAY] 3 macrovariables created
2
3    %array(r[3] (4 5 6))
NOTE:[ARRAY] 3 macrovariables created
4
5    %put _user_;
GLOBAL Q2 E
GLOBAL Q3 F
GLOBAL Q4 G
GLOBAL QHBOUND 4
GLOBAL QLBOUND 2
GLOBAL QN 3
GLOBAL R1 4
GLOBAL R2 5
GLOBAL R3 6
GLOBAL RHBOUND 3
GLOBAL RLBOUND 1
GLOBAL RN 3
6    %concatArrays(q, r);
7    %put _user_;
GLOBAL Q2 E
```

```
GLOBAL Q3 F
GLOBAL Q4 G
GLOBAL Q5 4
GLOBAL Q6 5
GLOBAL Q7 6
GLOBAL QHBOUND 7
GLOBAL QLBOUND 2
GLOBAL QN 6
```

In both cases we can also notice how indexes increase is automatically maintained.

The %do_over() macro.

We already saw how the `macarray=` parameter of the `%array()` macro makes do-looping over macro array elements simpler. A macro we discuss in this subsection, the `%do_over()` macro, is a helper macro for the looping process itself. It allows for the looping over a macro array elements without the necessity of "putting" the do-loop inside an extra/external macro. As the result the `%do_over()` generates *plain text* which is then "used" (of course the result depends on the context into which the text is pushed in the code, see the next two examples). Also when executed, the `%do_over()` macro *assumes* that the `%array()` macro was run with the `macarray=` parameter set to `Y`.

For example the following:

code: silly %do_over() looping

```
%array(ds=sashelp.class(where=(age=12)), vars=name, q=2, macarray=Y)

%do_over(name)

%put _user_;
```

will rise an error message in the log:

the log - error

```
1    %array(ds=sashelp.class(where=(age=12)), vars=name, q=2, macarray=Y)
NOTE:[ARRAY] 5 macrovariables created
2
3    %do_over(name)
NOTE: Line generated by the macro function "UNQUOTE".
1       "James"
        -------
        180
ERROR 180-322: Statement is not valid or it is used out of proper order.

4
5    %put _user_;
GLOBAL NAME1 "James"
GLOBAL NAME2 "Jane"
GLOBAL NAME3 "John"
GLOBAL NAME4 "Louise"
GLOBAL NAME5 "Robert"
GLOBAL NAMEHBOUND 5
```

```
GLOBAL NAMELBOUND 1
GLOBAL NAMEN 5
```

because SAS does not know how to interpret the `"James" "Jane" ... "Robert"` plain text string in the open code.

But when we add the `%put` statement or place the `%do_over()` result in proper context (e.g. a data step code) it works as intended:

code: wise %do_over() looping

```
%array(ds=sashelp.class(where=(age=12)), vars=name, q=2, macarray=Y)

%put %do_over(name);

data test;
  set sashelp.class;
  where name in (%do_over(name));
run;
```

which can be confirmed in log printout:

the log - good looping

```
1    %array(ds=sashelp.class(where=(age=12)), vars=name, q=2, macarray=Y)
NOTE:[ARRAY] 5 macrovariables created
2
3    %put %do_over(name);
"James" "Jane" "John" "Louise" "Robert"
4
5    data test;
6      set sashelp.class;
7      where name in (%do_over(name));
8    run;

NOTE: There were 5 observations read from the data set SASHELP.CLASS.
      WHERE name in ('James', 'Jane', 'John', 'Louise', 'Robert');
NOTE: The data set WORK.TEST has 5 observations and 5 variables.
```

As we can see from the examples, when the `%do_over()` is executed in its most basic form, i.e., with only a macro array name as the `array` parameter value, the result is a text string listing all elements of the macro array separated by space character.

The basic form, which uses only a macro array name, has a lot of practical use cases but, when we use additional key-value parameters then the `%do_over()` reveals its full potential. The first key-value parameter is the `between=` parameter, it provides a "separator" string which is inserted between consecutive iterations. The default is space character.

The following example uses the `"" ""` text (double quote, space, and double quote) for the `between=` and does not use the `q=2` in the MVA definition, plus the `%do_over()` is "wrapped" in double quotes:

code: %do_over() looping with "between="

```
%array(ds=sashelp.class(where=(age=12)), vars=name, macarray=Y)

%put "%do_over(name, between=" ")";
```

Log shows the following:

the log - "between="

```
1    %array(ds=sashelp.class(where=(age=12)), vars=name, macarray=Y)
NOTE:[ARRAY] 5 macrovariables created
2
3    %put "%do_over(name, between=" ")" ;
"James" "Jane" "John" "Louise" "Robert"
```

We can observe that this and the previous example produce the same result.

Next key-value parameter is the `phrase=`, this parameter keeps the text of a phrase which is executed in each iteration of the `%do_over()` (see examples). The default value of the parameter, which results in printing all elements of a macro array, is the following phrase of text: "`%nrstr(%&array(&_i_.))`". The macro variable `&_i_.` is an implicit internal iterator of the `%do_over()` macro, `&_i_.` can be used inside phrases you want to loop by the `%do_over()` macro (use of `&_i_.` instead `&i.` was dictated by desire of similarity to 4GL's "`do over Array;`" statement and its default iterator `_i_`). There is a requirement for phrases, when calling to `&_i_.` or "`macarray=Y` generated macros", that they *have to be* "enveloped" inside the `%NRSTR()` macro quoting function! Of course when the `%NRSTR()` is used its masking rules, e.g., for brackets or apostrophes, has to be kept. So in practice, when we are thinking: "`phrase=<...code...>`" it is better to think: "`phrase=%NRSTR(<...code...>)`".

Now, let's look at some examples. In the first one we want to split a single data set into separate data sets based on a variable's values:

code: using "phrase=" parameter, ex. 1

```
%array(ds=sashelp.class, vars=age|, macarray=Y)

options mprint;
data
  %do_over(age, phrase=%nrstr(data_%age(&_i_.)))
;
  set sashelp.class;

  select(age);
    %do_over(age, phrase=%nrstr(
     when (%age(&_i_.)) output data_%age(&_i_.);
     ))
    otherwise put "unknown Age value!";
  end;
run;
```

We are using the `%do_over()` macro twice, once to create a list of data sets, second to construct the `select(); ... end;` statement. With the help of `options mprint;` we can see that in the log:

the log - using "phrase=" parameter, ex. 1

```
1    %array(ds=sashelp.class, vars=age|, macarray=Y)
NOTE:[ARRAY] 6 macrovariables created
2
3    options mprint;
4    data
5      %do_over(age, phrase=%nrstr(data_%age(&_i_.)))
MPRINT(DO_OVER):   data_15
MPRINT(DO_OVER):   data_12
```

```
MPRINT(DO_OVER):   data_16
MPRINT(DO_OVER):   data_13
MPRINT(DO_OVER):   data_14
MPRINT(DO_OVER):   data_11
6     ;
7       set sashelp.class;
8
9       select(age);
10        %do_over(age, phrase=%nrstr(
11         when (%age(&_i_.)) output data_%age(&_i_.);
12         ))
MPRINT(DO_OVER):   when (15) output data_15;
MPRINT(DO_OVER):   when (12) output data_12;
MPRINT(DO_OVER):   when (16) output data_16;
MPRINT(DO_OVER):   when (13) output data_13;
MPRINT(DO_OVER):   when (14) output data_14;
MPRINT(DO_OVER):   when (11) output data_11;
13        otherwise put "unknown Age value!";
14      end;
15    run;
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.DATA_15 has 4 observations and 5 variables.
NOTE: The data set WORK.DATA_12 has 5 observations and 5 variables.
NOTE: The data set WORK.DATA_16 has 1 observations and 5 variables.
NOTE: The data set WORK.DATA_13 has 3 observations and 5 variables.
NOTE: The data set WORK.DATA_14 has 4 observations and 5 variables.
NOTE: The data set WORK.DATA_11 has 2 observations and 5 variables.
```

In the second example we want to run a user defined macro over several consecutive dates as that macro's parameter value:

code: using "phrase=" parameter, ex. 2

```
%array(date[4], function=202300 + _i_, macarray=Y)

%put %do_over(date);

%macro someMacro4(period);
  data _null_;
    put "Running macro for &period.";
  run;
%mend someMacro4;

%do_over(date, phrase=%nrstr(
  %someMacro4(%date(&_i_.))
))
```

With help of the `function=` parameter in the `%array()` macro we are creating a list of dates (in the "YYYYMM" form) which will be passed as arguments in `%do_over()` in order to use them as parameters in the call to `%someMacro4()` macro:

the log - using "phrase=" parameter, ex. 2

```
1    %array(date[4], function=202300 + _i_, macarray=Y)
NOTE:[ARRAY] 4 macrovariables created
2
3    %put %do_over(date);
202301 202302 202303 202304
4
5    %macro someMacro4(period);
6      data _null_;
7        put "Running macro for &period.";
8      run;
9    %mend someMacro4;
10
11   %do_over(date, phrase=%nrstr(
12     %someMacro4(%date(&_i_.))
13   ))

Running macro for 202301
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

Running macro for 202302
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

Running macro for 202303
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

Running macro for 202304
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Third example shows that inside the `%do_over()`'s phrases we can use *multiple macro arrays* at once:

code: using "phrase=" parameter, ex. 3

```
%array(ds=sashelp.class(obs=6), vars=name age, macarray=Y)

%do_over(name, phrase=%nrstr(
  %put &_i_.. %name(&_i_.) has %age(&_i_.) years.;
))
```

If two or more macro arrays have the same number of elements and "aligned" indexes, then they can be used in the `phrase=` parameter text. One of those macros should be used as the first parameter of the `%do_over()` to be a "driving" macro array (like the `name` MVA in the example).

the log - using "phrase=" parameter, ex. 3

```
1    %array(ds=sashelp.class(obs=6), vars=name age, macarray=Y)
NOTE:[ARRAY] 12 macrovariables created
2
3    %do_over(name, phrase=%nrstr(
4      %put &_i_.. %name(&_i_.) has %age(&_i_.) years.;
5    ))
1. Alfred has 14 years.
2. Alice has 13 years.
3. Barbara has 13 years.
4. Carol has 14 years.
5. Henry has 14 years.
6. James has 12 years.
```

The last parameter of the `%do_over()` is `which=`, this one expects a space separated list of "indexes lists" over which the `%do_over()` will iterate. The idea behind this parameter is to allow subselection of a macro array elements. A "indexes list" is a construction of the form `start<:end<:by>>`, examples of such "indexes lists" are: `1:10:1`, `1:10`, `1:10:2`, `1`, `10`, or even `10:1:-1`. The `which=` allows multiple "indexes lists" but sometimes single one is "better" i.e., it is better if multiple index lists can be condensed to a single index list. For example, single "index list" of the form: `1:5` is equivalent to five "index lists": `1 2 3 4 5`, but use of the first one is more efficient under the hood. There are some technical requirements about "indexes list": 1) default value of `by` is `1`, 2) a single "index list" cannot have spaces in its definition, 3) there a two special symbols: `H` and `L` which refers to higher and lower index values. Interesting example where `H`, `L`, and `-1` are used is to loop over a macro array backward:

code: using "which=" parameter

```
%array(test[*] x01-x06, vnames=Y, macarray=Y)

%put 1) %do_over(test);
%put 2) %do_over(test, which=H:L:-1);
```

The first loop from one to six, the second from six to one:

the log - using "which=" parameter

```
1    %array(test[*] x01-x06, vnames=Y, macarray=Y)
NOTE:[ARRAY] 6 macrovariables created
2
3    %put 1) %do_over(test);
1) x01 x02 x03 x04 x05 x06
4    %put 2) %do_over(test, which=H:L:-1);
2) x06 x05 x04 x03 x02 x01
```

Log printout from yet another example:

the log - using "which=" parameter, ex. 2

```
1    %array(test[*] x01-x99, vnames=Y, macarray=Y)
NOTE:[ARRAY] 99 macrovariables created
2
3    %put %do_over(test, which=L:3 97:H);
x01 x02 x03 x97 x98 x99
```

The %do_overN() and the %make_do_over() macros.

The %do_over2() and %do_over3() macros are utility macros which "extend" functionality of the %do_over().

The %do_over() macro allows to loop over a macro array. The %do_over2() macro allows to loop over *two* arrays by "loop-in-loop" approach, which gives us a Cartesian product of two arrays. The %do_over3() macro allows to loop over *three* arrays by "loop-in-loop-in-loop" approach, which gives us a Cartesian product of three arrays. Two following examples present use cases.

The first example shows how to run a macro with all possible combinations of input parameters values.

code: %do_over2()calling a two parameters macro

```
options nofullstimer nostimer; /* for shorter log */
%array(alpha[3] $ ("A" "B" "C"), macarray=Y)
%array( beta[3]   ( 1   2   3 ), macarray=Y)

%macro letsPlay(x,y);
  data &x&y;
    x="&x.";
    y= &y. ;
    put x= y=;
  run;
%mend letsPlay;

%do_over2(alpha, beta
  ,phrase = %NRSTR(
    %letsPlay(%alpha(&_I_.), %beta(&_J_))
  ))
```

The log shows nine data sets created:

the log - nine data sets from %do_over2()

```
1    options nofullstimer nostimer; /* for shorter log */
2    %array(alpha[3] $ ("A" "B" "C"), macarray=Y)
NOTE:[ARRAY] 3 macrovariables created
3    %array( beta[3]   ( 1   2   3 ), macarray=Y)
NOTE:[ARRAY] 3 macrovariables created
4
5    %macro letsPlay(x,y);
6      data &x&y;
7        x="&x.";
8        y= &y. ;
9        put x= y=;
10     run;
11   %mend letsPlay;
12
13   %do_over2(alpha, beta
14     ,phrase = %NRSTR(
15       %letsPlay(%alpha(&_I_.), %beta(&_J_))
16     ))
19
20 x=A y=1
```

```
NOTE: The data set WORK.A1 has 1 observations and 2 variables.

x=A y=2
NOTE: The data set WORK.A2 has 1 observations and 2 variables.

x=A y=3
NOTE: The data set WORK.A3 has 1 observations and 2 variables.

x=B y=1
NOTE: The data set WORK.B1 has 1 observations and 2 variables.

x=B y=2
NOTE: The data set WORK.B2 has 1 observations and 2 variables.

x=B y=3
NOTE: The data set WORK.B3 has 1 observations and 2 variables.

x=C y=1
NOTE: The data set WORK.C1 has 1 observations and 2 variables.

x=C y=2
NOTE: The data set WORK.C2 has 1 observations and 2 variables.

x=C y=3
NOTE: The data set WORK.C3 has 1 observations and 2 variables.
```

The second example shows Cartesian product of a single macro array with itself and itself... so three times.

code: %do_over3()producing Cartesian product over macro array A

```
%array(a[2] (0 1), macarray=Y)

%do_over3(a, a, a
, phrase = %NRSTR(
  %put sum(%a(&_I_.),%a(&_J_),%a(&_K_))=%sysevalf(%a(&_I_.)+%a(&_J_)+%a(&_K_));
))
```

In the log we can see:

the log - all possible sums

```
1    %array(a[2] (0 1), macarray=Y)
NOTE:[ARRAY] 2 macrovariables created
2
3    %do_over3(a, a, a
4    , phrase = %NRSTR(
5      %put sum(%a(&_I_.),%a(&_J_),%a(&_K_))=%sysevalf(%a(&_I_.)+%a(&_J_)+%a(&_K_));
6    ))
sum(0,0,0)=0
sum(0,0,1)=1
sum(0,1,0)=1
sum(0,1,1)=2
```

```
sum(1,0,0)=1
sum(1,0,1)=2
sum(1,1,0)=2
sum(1,1,1)=3
```

As we can notice both macros have dedicated internal looping variables. For the `%do_over2()` macro they are `_I_` and `_J_`, and for The `%do_over3()` macro we also have `_K_`. The `%do_over*()` macros have *no* `which=` parameter for them, only `phrase=` and `between=` are available.

With fairly high probability we can say that 99% of practical situations of looping over macro arrays can be done with `%do_over()`(80%), `%do_over2()`(13%), and with `%do_over3()`(6%), but there is still this 1% of "special situations" where a "`do_over` of higher order" may be required (i.e. greater than 3 nested loops). That is the reason why the `%make_do_over()` macro was implemented. Since it is rather "special case" situation we only look at logs from two small examples.

the log - size must be greater than 3

```
1    %make_do_over(2);
NOTE:[MAKE_DO_OVER] NO MACRO GENERATED FOR SIZE = 2
     [MAKE_DO_OVER] SIZE must be greater than 3!!!
```

the log - loop of size five

```
1    %make_do_over(5);

NOTE: The file _******_ is:
      Filename=********************************,
      RECFM=V,LRECL=512,File Size (bytes)=0,
      Last Modified=19May2024:12:34:56,
      Create Time=19May2024:12:34:56
NOTE: 41 records were written to the file _******_.
      The minimum record length was 1.
      The maximum record length was 64.
NOTE: Fileref _******_ has been deassigned.
43
44   %array(a5_[2] (0 1), macarray=Y)
NOTE:[ARRAY] 2 macrovariables created
45
46   %do_over5(a5_, a5_, a5_, a5_, a5_
47   ,phrase = %NRSTR(
48     %put (%a5_(&_I1_.),%a5_(&_I2_),%a5_(&_I3_),%a5_(&_I4_),%a5_(&_I5_));
49   ))
(0,0,0,0,0)
(0,0,0,0,1)
...
...
...
(1,1,1,1,0)
(1,1,1,1,1)
```

What is worth mentioning, the loop index variables are in this case `_I1_`, `_I2_`, `_I3_`, etc. All technical details are described in *the documentation of the macro*. As mentioned earlier, all the documentation is available at `macroArray` package's repository.

The %zipArrays() and the %QzipArrays() macros.

Two final macros from the macro arrays suite in the `macroArray` package are dedicated to execute operations and functions on respective elements of two macro arrays and produce a new macro array with results of processing. The only difference between them is that the `%QzipArrays()`returns macro quoted results and `%zipArrays()` does not. The "zip" in the macro name has nothing to do with data compression, its rather a reference to a "zipper", where *corresponding* pairs of teeth from the left and right row are "meshed" together. When one of macro arrays has fewer elements than the other then elements of the shorter are, by default, "reused" starting from the beginning. This behavior can be altered, to produce Cartesian product or stop at shorter, with help of the `reuse=` parameter.

Two positional parameters, which are required, are names of macro arrays. The default function operating on corresponding elements is the `cat()` function, the name of newly created macro array is concocted from names of input macro arrays. Both the function and the name can be altered, respectively, by the `function=` and the `result=` parameter. Also the `macarray=` parameter (known from the `%array()` macro) is available.

The following example creates new macro array `ab` with concatenated values of macro arrays `a` and `b` (values of `b` are reused):

code: simple zip of two macro arrays

```
%array(a[4] X Y Z T, vnames=Y)
%array(b[2] (1 2))

%put _user_;
%zipArrays(a, b);
%put _user_;
```

The log reveals the following information:

the log - new macro array AB highlighted

```
1    %array(a[4] X Y Z T, vnames=Y)
NOTE:[ARRAY] 4 macrovariables created
2    %array(b[2] (1 2))
NOTE:[ARRAY] 2 macrovariables created
3
4    %put _user_;
GLOBAL A1 X
GLOBAL A2 Y
GLOBAL A3 Z
GLOBAL A4 T
GLOBAL AHBOUND 4
GLOBAL ALBOUND 1
GLOBAL AN 4
GLOBAL B1 1
GLOBAL B2 2
GLOBAL BHBOUND 2
GLOBAL BLBOUND 1
GLOBAL BN 2
5    %zipArrays(a, b);
NOTE:[ZIPARRAYS] 4 macrovariables created
6    %put _user_;
GLOBAL A1 X
```

```
GLOBAL A2 Y
GLOBAL A3 Z
GLOBAL A4 T
GLOBAL AB1 X1
GLOBAL AB2 Y2
GLOBAL AB3 Z1
GLOBAL AB4 T2
GLOBAL ABHBOUND 4
GLOBAL ABLBOUND 1
GLOBAL ABN 4
GLOBAL AHBOUND 4
GLOBAL ALBOUND 1
GLOBAL AN 4
GLOBAL B1 1
GLOBAL B2 2
GLOBAL BHBOUND 2
GLOBAL BLBOUND 1
GLOBAL BN 2
```

This example creates new macro array and utilizes parameters presented above (new macro array is Cartesian product in this case):

code: zip of two macro arrays with parameters

```
%array(a[2] (100 200))
%array(b[2] (3 4))

%put _user_;
%zipArrays(a, b
, result=NEW_ARR
, function=SUM
, reuse=CP
, macarray=Y
)
%put _user_;

%put %do_over(NEW_ARR);
```

The log shows everything, even additional note for the `macarray=` parameter:

the log - zip of two macro arrays with parameters

```
1    %array(a[2] (100 200))
NOTE:[ARRAY] 2 macrovariables created
2    %array(b[2] (3 4))
NOTE:[ARRAY] 2 macrovariables created
3
4    %put _user_;
GLOBAL A1 100
GLOBAL A2 200
GLOBAL AHBOUND 2
GLOBAL ALBOUND 1
GLOBAL AN 2
```

```
12 GLOBAL B1 3
13 GLOBAL B2 4
14 GLOBAL BHBOUND 2
15 GLOBAL BLBOUND 1
16 GLOBAL BN 2
17 5    %zipArrays(a, b
18 6    , result=NEW_ARR
19 7    , function=SUM
20 8    , reuse=CP
21 9    , macarray=Y
22 10   )
23 NOTE:[ZIPARRAYS] 4 macrovariables created
24
25 NOTE: When macarray= parameter is active the zipArrays macro
26       cannot be called within the %put statement.
27 NOTE: Execution like: %put %zipArrays(..., macarray=Y) will
28       result with an e.r.r.o.r.
29 11   %put _user_;
30 GLOBAL A1 100
31 GLOBAL A2 200
32 ...
33 GLOBAL BLBOUND 1
34 GLOBAL BN 2
35 GLOBAL NEW_ARR1 103
36 GLOBAL NEW_ARR2 104
37 GLOBAL NEW_ARR3 203
38 GLOBAL NEW_ARR4 204
39 GLOBAL NEW_ARRHBOUND 4
40 GLOBAL NEW_ARRLBOUND 1
41 GLOBAL NEW_ARRN 4
42 12
43 13   %put %do_over(NEW_ARR);
44 103 104 203 204
```

See the documentation where all remaining details about additional parameters, not mentioned here, are given. Also many more examples are presented there.

The `%mcDictionary()` macro.

Regarding the list of questions that were presented at the beginning of this paper that led to the development of the `macroArray` package, one more question can be added: Does the "`suffix`" (array's index) have to always be an integer? Can we, instead writing for example `%myArray(1)`, write `%myDictionary(Bart)` and get the value, let's say "`180cm`", extracted for key "`Bart`"? Yes, the `macroArray` package also provides such functionality. Let's talk about *macro dictionaries*.

Macro dictionary (aka MD) is a concept similar to the macro variable array, but instead "integer-value" pairs like MVAs have, it allows for a "key-value" relations between macro variables values. The "key" in this case can be arbitrary text string associated with a value and the value is retrieved from macro dictionary by providing the key. In the essence, the MD behavior is similar to the way formats or hash tables behaves (though the mechanism under the hood is different). Worth to mention is that a macro dictionary works according to the "one key - one value" assumption.

The macroArray package facilitates macro dictionaries by the %mcDictionary() macro. A dictionary is created by calling the %mcDictionary() macro with a name to be used by the dictionary. The naming convention is "SAS standard" with the respect to the following restrictions: cannot be empty, cannot be underscore, cannot be longer than 13 symbols. First, we create an empty macro dictionary named myDict:

code: simple (and empty) dictionary

```
%mcDictionary(myDict)
```

The log does not show us anything interesting but it does not mean nothing happened. In fact quite a lot has happened. Under the hood a macro named %myDict() was generated, similar way when we used macarray= parameter in the %array() macro, which instantiate the MD and its use. The %myDict() macro, and also any other macro created by the %mcDictionary() macro, has three parameters. One positional and two "key-value" type. The first, positional parameter named method, is to tell the %myDict() macro what work to execute in particular under the hood. It accepts the following list of values: LIST [L], ADD [A], FIND [F], CHECK [C], DEL [D], and CLEAR (in square brackets are aliases). We discuss them in a moment, but let's finish describing parameters first. The "key-value" parameters are: key= and data= and whether they are used or not depends on the value of the method parameter.

Let's talk about the method parameter values (also referred simply as "methods"). When method is empty then a table with list of accepted values is printed in the log.

The first method we discuss is the LIST [L], which makes %myDict() to print out in the log all its elements (keys, values, and hash digest associated with key). In our case since the %myDict() has no elements yet, the printout will be just:

the log - list of empty dictionary

```
1    %myDict(List)

Content of the dictionary:

--------------------------
```

To populate our MD with new data we use the ADD [A] method (this is one of two ways of populating MD with data). When we use ADD [A] the key= provides value for the key in the MD and the data= provides value for the data portion in the MD. For example, the following code adds three entries to the %myDict() MD we created:

code: adding data to MD

```
%myDict(ADD,key=A,data=I)
%myDict(A  ,key=B,data=<3)
%myDict(A  ,key=C,data=SAS)
%myDict(L)
```

And in the log we can see:

the log - three entries and listing

```
1    %myDict(ADD,key=A,data=I)
2    %myDict(A  ,key=B,data=<3)
3    %myDict(A  ,key=C,data=SAS)
4    %myDict(L)

Content of the dictionary:

key=C
```

```
hash=0D61F8370CAD1D41
val=SAS

key=A
hash=7FC56270E7A70FA8
val=I

key=B
hash=9D5ED678FE57BCCA
val=<3

--------------------------
```

As we can see, the process of adding data to the MD does not reveal too much but yields of the process can be verified by the `LIST` [`L`] method.

When the MD is not empty we can try to use the data. This can be done in two ways. The first, we can just check if a given key exists in the MD, the `CHECK` [`C`] method does that. For example, the following code returns zeros and ones of existing and non existing keys:

code: checking keys

```
%put
 A:%myDict(CHECK,key=A)
 B:%myDict(C,key=B)
 C:%myDict(C,key=C)
 D:%myDict(C,key=D)
;
```

The log shows:

the log - checking keys

```
A:1  B:1  C:1  D:0
```

The second, we can for a given key extract associated data portion, the `FIND` [`F`] method does that. For example, to get: "I ♡ SAS" in the log we run:

code: collecting data values

```
%put
"%myDict(FIND,key=A)
 %myDict(F,key=B)
 %myDict(F,key=C)
 %myDict(F,key=D)"
;
```

The log shows (mind there is a space character at the beginning of lines 3, 4, and 5 in the code):

the log - collecting data values

```
"I <3 SAS "
```

As we can see the value of not existing key "D" is empty (the space is the one from between the bracket and the percent in `...=C) %myD...`).

We discuss two final methods in one thread since both are dedicated to removing data from a MD. The methods are: `DEL` [`D`], and `CLEAR`. The difference between them is that the `DEL` [`D`] removes data for only

a particular key (so `key=` parameter is needed) and the `CLEAR` just deletes them all and `CLEAR` does not have an alias. Let's see it on the following example:

code: deleting entries from MD

```
%myDict(DEL,key=B)
%myDict(L)

%myDict(CLEAR)
%myDict(L)
```

In the log we can see:

the log - deleting entries from MD

```
1     %myDict(DEL,key=B)
2     %myDict(L)

Content of the dictionary:

key=C
hash=0D61F8370CAD1D41
val=SAS

key=A
hash=7FC56270E7A70FA8
val=I

---------------------------
3
4     %myDict(CLEAR)
5     %myDict(L)

Content of the dictionary:

---------------------------
```

In both cases, under the hood macro variables of the form `myDict_****************_K` and `myDict_****************_V` are looked up, the "`****************`" indicates a sixteen symbols of alphanumeric hash digest of the key. But for the `DEL [D]` method only "that one" is deleted, while the `CLEAR` methods removes them all. As a side note, by looking at the internal representation of MD variables, we now know why the naming convention limits the name to 13 characters (*32 - (16+3)*).

When we were discussing the `ADD` method a second method of populating MD with data was mentioned. The method, similarly as in the macro arrays case is based on taking data from the data set. To use data set we have to indicate the fact when we execute the `%mcDictionary()` macro. Three dedicated parameters: `ds=`, `k=`, and `d=` can be use for the process. The `ds=` indicates data set, the `k=` indicates key variable (one variable!) in the data set, and the `d=` indicates data portion variable (also one variable!) in the data set. Default values for `k=` and `d=` are "Key" and "Data" so if the data set contains such variables and we want use them we can skip `k=` and `d=`. The following code present an example:

code: populating MD from data set

```
data work.have;
  input kVar :$1. dVar :$3.;
cards;
```

```
A I
B <3
C SAS
;
run;

%mcDictionary(myDSdict, DCL, ds=work.have, k=kVar, d=dVar)

%myDSdict(L)
```

Notes in the log confirm everything:

the log - populating MD from data set

```
1    data work.have;
2      input kVar :$1. dVar :$3.;
3    cards;
NOTE: The data set WORK.HAVE has 3 observations and 2 variables.
...
7    ;
8    run;
9
10   %mcDictionary(myDSdict, DCL, ds=work.have, k=kVar, d=dVar)
NOTE:[MCDICTIONARY] Populating dictionary myDSdict.
NOTE:[MCDICTIONARY] ----------------------------
11
12   %myDSdict(L)

Content of the dictionary:

key=C
hash=0D61F8370CAD1D41
val=SAS

key=A
hash=7FC56270E7A70FA8
val=I

key=B
hash=9D5ED678FE57BCCA
val=<3

--------------------------
```

Except the parameters mentioned above, one more thing can be observed in call to `%mcDictionary(..., DCL,...)` macro. The `DCL` value (also `DECLARE` or just null) in the second parameter tells the `%mcDictionary()` macro that either a macro dictionary is to be created (when parameter is `DCL`) or a macro dictionary is to be deleted (when parameter is `DELETE`). If the `DELETE` is used not only MD's macro variables are deleted but the MD macro itself is removed from the SAS session.

EXAMPLE

In the last section we will take a look at some more ”advanced” examples that illustrate the use of both macro arrays and macro dictionaries.

As the first step we will prepare ”driving” data sets:

code: creating "driving" data sets

```
/* project driving data */
/* data set with list of functions to "run over data" */
data work.functions;
  input fName $12.;
cards;
sum
mean
median
min
max
nmiss
std
range
stderr
var
;
run;

/* data set with project metadata */
data work.projectMetadata;
  infile cards dsd dlm=",";
  input key :$16. data :$128.;
cards;
ID,ABC-123-XYZ
TITLE,Use case of the MacroArray package
PATH,/path/to/study/data
INDATASET,sashelp.cars
OUTDATASET,work.results
VARIABLE,invoice
GROUPBY,origin
STARTDT,2020-01-01
ENDDT,2024-12-31
;
run;
```

The log confirms data sets creation:

the log - creating "driving" data sets

```
1    /* project driving data */
2    /* data set with list of functions to "run over data" */
3    data work.functions;
4      input fName $12.;
5    cards;
6
```

```
NOTE: The data set WORK.FUNCTIONS has 10 observations and 1 variables.
...

16   ;
17   run;
18
19   /* data set with project metadata */
20   data work.projectMetadata;
21     infile cards dsd dlm=",";
22     input key :$16. data :$128.;
23   cards;

NOTE: The data set WORK.PROJECTMETADATA has 9 observations and 2 variables.
...

33   ;
34   run;
```

The driving data allows us to execute the "project code" and the use of macro array and macro dictionary makes the project code only "data driven". Here is the code:

code: data driven project

```
/* project code */
/* create macro array FN and macro dictionary PRJ */
%array(ds=work.functions, vars=fName#fN, macarray=Y)
%mcDictionary(prj, DCL, DS=work.projectMetadata)

%put _user_;

title1 "Title: %prj(F,key=TITLE)";
title2 "Project %prj(F,key=ID), located at: %prj(F,key=PATH)";
title3 "starts %prj(F,key=STARTDT) and ends %prj(F,key=ENDDT)";
footnote1 "Input data set: %prj(F,key=INDATASET)";
footnote2 "Output data set: %prj(F,key=OUTDATASET)";
footnote3 "Analyzed variable: %prj(F,key=VARIABLE)";
/* check if the grouping variable exists */
footnote4
  %if %prj(C,key=GROUPBY) %then
    %do; "Analysis in groups by: %prj(F,key=GROUPBY)" %end;
;

/* aggregate data */
Proc SQL;
  create table %prj(F,key=OUTDATASET) as
  select
    /* check if the grouping variable exists */
    %if %prj(C,key=GROUPBY) %then %do; %prj(F,key=GROUPBY), %end;
    /* loop over aggregating functions */
    %do_over(fN
      ,phrase=%NRSTR(
        /* apply function to analysisVariable
```

```
            and name the result "analysisVariable_functionName" */
          %fN(&_i_.)(%prj(F,key=VARIABLE)) as %prj(F,key=VARIABLE)_%fN(&_i_.)
         )
        ,between=%str(,)
        )

  from
    %prj(F,key=INDATASET)

  /* check if the grouping variable exists */
  %if %prj(C,key=GROUPBY) %then
    %do;
      group by
        %prj(F,key=GROUPBY)
    %end;
  ;
Quit;

/* print data */
proc print data = %prj(F,key=OUTDATASET) ;
run;

title;
footnote;

/* end of project code */
```

The log from the process is the following:

the log - data driven project

```
1    /* project code */
2    /* create macro array FN and macro dictionary PRJ */
3    %array(ds=work.functions, vars=fName#fN, macarray=Y)
NOTE:[ARRAY] 10 macrovariables created
4    %mcDictionary(prj, DCL, DS=work.projectMetadata)
NOTE:[MCDICTIONARY] Populating dictionary prj.
NOTE:[MCDICTIONARY] ----------------------------
5
6    %put _user_;
GLOBAL FN1 sum
GLOBAL FN2 mean
GLOBAL FN3 median
GLOBAL FN4 min
GLOBAL FN5 max
GLOBAL FN6 nmiss
GLOBAL FN7 std
GLOBAL FN8 range
GLOBAL FN9 stderr
GLOBAL FN10 var
GLOBAL FNHBOUND 10
GLOBAL FNLBOUND 1
```

```
22 GLOBAL FNN 10
23 GLOBAL PRJ_5FFB5F0D0DE78321_K PATH
24 GLOBAL PRJ_5FFB5F0D0DE78321_V /path/to/study/data
25 GLOBAL PRJ_6F9DCCD85B2E0786_K TITLE
26 GLOBAL PRJ_6F9DCCD85B2E0786_V Use case of the MacroArray package
27 GLOBAL PRJ_815250411C5E7F5E_K STARTDT
28 GLOBAL PRJ_815250411C5E7F5E_V 2020-01-01
29 GLOBAL PRJ_97C6507873878CEA_K GROUPBY
30 GLOBAL PRJ_97C6507873878CEA_V origin
31 GLOBAL PRJ_9B54A6B75069B38A_K OUTDATASET
32 GLOBAL PRJ_9B54A6B75069B38A_V work.results
33 GLOBAL PRJ_AC8C59A0734680FB_K ENDDT
34 GLOBAL PRJ_AC8C59A0734680FB_V 2024-12-31
35 GLOBAL PRJ_B718ADEC73E04CE3_K ID
36 GLOBAL PRJ_B718ADEC73E04CE3_V ABC-123-XYZ
37 GLOBAL PRJ_D519071E56F75CF8_K INDATASET
38 GLOBAL PRJ_D519071E56F75CF8_V sashelp.cars
39 GLOBAL PRJ_E61AD9B2553A293B_K VARIABLE
40 GLOBAL PRJ_E61AD9B2553A293B_V invoice
41 GLOBAL PRJ_KEYSNUM 0
42
43 7
44 8    title1 "Title: %prj(F,key=TITLE)";
45 9    title2 "Project %prj(F,key=ID), located at: %prj(F,key=PATH)";
46 10   title3 "starts %prj(F,key=STARTDT) and ends %prj(F,key=ENDDT)";
47 11   footnote1 "Input data set: %prj(F,key=INDATASET)";
48 12   footnote2 "Output data set: %prj(F,key=OUTDATASET)";
49 13   footnote3 "Analyzed variable: %prj(F,key=VARIABLE)";
50 14   /* check if the grouping variable exists */
51 15   footnote4
52 16     %if %prj(C,key=GROUPBY) %then
53 17       %do; "Analysis in groups by %prj(F,key=GROUPBY)" %end;
54 18   ;
55 19
56 20   /* aggregate data */
57 21   Proc SQL;
58 22     create table %prj(F,key=OUTDATASET) as
59 MPRINT(PRJ):   work.results
60 23     select
61 24       /* check if the grouping variable exists */
62 25       %if %prj(C,key=GROUPBY) %then %do; %prj(F,key=GROUPBY), %end;
63 MPRINT(PRJ):   origin
64 26       /* loop over aggregating functions */
65 27       %do_over(fN
66 28         ,phrase=%NRSTR(
67 29           /* apply function to analysisVariable
68 30              and name the result "analysisVariable_functionName" */
69 31           %fN(&_i_.)(%prj(F,key=VARIABLE)) as %prj(F,key=VARIABLE)_%fN(&_i_.)
70 32          )
```

```
33         ,between=%str(,)
34         )
MPRINT(DO_OVER):   sum(invoice) as invoice_sum,
MPRINT(DO_OVER):  mean(invoice) as invoice_mean,
MPRINT(DO_OVER):  median(invoice) as invoice_median,
MPRINT(DO_OVER):  min(invoice) as invoice_min,
MPRINT(DO_OVER):  max(invoice) as invoice_max,
MPRINT(DO_OVER):  nmiss(invoice) as invoice_nmiss,
MPRINT(DO_OVER):  std(invoice) as invoice_std,
MPRINT(DO_OVER):  range(invoice) as invoice_range,
MPRINT(DO_OVER):  stderr(invoice) as invoice_stderr,
MPRINT(DO_OVER):  var(invoice) as invoice_var
35
36     from
37       %prj(F,key=INDATASET)
MPRINT(PRJ):   sashelp.cars
38
39     /* check if the grouping variable exists */
40     %if %prj(C,key=GROUPBY) %then
41       %do;
42         group by
43           %prj(F,key=GROUPBY)
MPRINT(PRJ):   origin
44       %end;
45     ;
NOTE: Table WORK.RESULTS created, with 3 rows and 11 columns.

46   Quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds


47
48   /* print data */
49   proc print data = %prj(F,key=OUTDATASET) ;
MPRINT(PRJ):   work.results
50   run;

NOTE: There were 3 observations read from the data set WORK.RESULTS.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


51
52   title;
53   footnote;
54
55   /* end of project code */
```

The code with help of macro array and macro dictionary generated from data sets is fully data driven. No changes in the code are needed, user need only modify driving data sets to either change list of aggregating functions or the project itself. And the output looks like this:

Title: Use case of the MacroArray package
Project ABC-123-XYZ, located at: /path/to/study/data
starts 2020-01-01 and ends 2024-12-31

Obs	Origin	invoice_sum	invoice_mean	invoice_median	invoice_min	invoice_max	invoice_nmiss	invoice_std	invoice_range	invoice_stderr	invoice_var
1	Asia	3571144	22602.18	20949.5	9875	79978	0	9842.98	70103	783.07	96884351.34
2	Europe	5460595	44395.08	37575.0	15437	173560	0	23080.37	158123	2081.09	532703428.01
3	USA	3814553	25949.34	23217.0	10319	74451	0	10518.72	64132	867.57	110643516.60

Input data set: sashelp.cars
Output data set: work.results
Analyzed variable: invoice
Analysis in groups by origin

CONCLUSION

In the article, we learned how to simplify the problem of creating SAS macro variable arrays with a help of the `macroArray` package. But before "the end" two more things require emphasis!

The first, for advanced SAS programmer, experienced in the art of "multi-ampersanding", who fully understand the process of indirect referencing, you can still *benefit from the package* by making your code easier to read and more straightforward in some cases. You can enjoy "syntactic sugar" of the `%array()` and `%mcDictionary()` macros the same way you enjoy the IF-subsetting (`if <conditio>;`) or the SUM-statement (`variable + <expression>;`).

The second, for a "junior" SAS programmer. The package is design to support inexperienced users walking their "multi-ampersands" path and the author encourages them to use the `macroArray` package for their benefit, but... at the same time the author *highly recommends* you to put an effort and attention in a deep understanding of "how does it work?". End then, when understood, to check the previous paragraph ;-)

The End

REFERENCES

[Widawski 2000] Mel Widawski, "Beginners Guide to Flexibility: Macro Variables",
WUSS Proceedings, 2000, https://www.lexjansen.com/wuss/2000/WUSS00084.pdf

[First 2001] Steven First, "Advanced Macro Topics",
SCSUG Proceedings, 2001, https://www.lexjansen.com/scsug/2001/SCSUG01131.pdf

[Widawski 2002] Mel Widawski, "Flexible Code the Easy Way: SAS Macro Variables",
WUSS Proceedings, 2002, https://www.lexjansen.com/wuss/2002/WUSS02124.pdf

[Fehd 2003] Ronald Fehd, "ARRAY: construction and usage of arrays of macro variables",
NESUG Proceedings, 2003, https://www.lexjansen.com/nesug/nesug03/cc/cc015.pdf

[Carpenter & Smith 2003] Arthur L. Carpenter & Richard O. Smith, "Data Management: Building a Dynamic Application",
MWSUG Proceedings, 2003, https://www.lexjansen.com/mwsug/2003/MWSUG03021.pdf

[Clay 2004] Ted Clay, "Macro Arrays Make %DO-Looping Easy",
WUSS Proceedings, 2004, https://www.lexjansen.com/wuss/2004/coders_corner/c_cc_macro_arrays_make_doloo.pdf

[Carpenter 2005] Arthur L. Carpenter, "Make 'em %LOCAL: Avoiding Macro Variable Collisions",
WUSS Proceedings, 2005, https://www.lexjansen.com/wuss/2005/sas_solutions/sol_make_em_local_avoiding.pdf

[Clay 2006] Ted Clay, "Five Easy (To Use) Macros",
PNWSUG Proceedings, 2006, https://www.lexjansen.com/pnwsug/2006/PN22TedClayFiveMacros.pdf

[Lavery 2007] Russell Lavery, "An Animated Guide: The Map of the SAS Macro Facility",
PHUSE Proceedings, 2007, https://www.lexjansen.com/phuse/2007/is/IS01.pdf

[Philp 2008] Stephen Philp, "SAS MACRO: Beyond the Basics",
SGF Proceedings, 2008, https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/045-2008.pdf

[Rosson 2009] Jennifer Rosson, "The Next Step with Macros – Double Ampersand Macros &&helpme&i",
WUSS Proceedings, 2009, https://www.lexjansen.com/wuss/2009/cod/COD-Rosson.pdf

[Russell & Tyndall 2010] Kevin Russell & Russ Tyndall, "SAS System Options: The True Heroes of Macro Debugging",
SGF Proceedings, 2010, https://support.sas.com/resources/papers/proceedings10/147-2010.pdf

[Gilsen 2011] Bruce Gilsen, "SAS Code and Macros: How They Interact",
SGF Proceedings, 2011, https://support.sas.com/resources/papers/proceedings11/243-2011.pdf

[Li 2012] Arthur Li, "Is Your Failed Macro Due To Misjudged "Timing"?",
SGF Proceedings, 2012, https://support.sas.com/resources/papers/proceedings12/228-2012.pdf

[Zender 2013] Cynthia L. Zender, "Macro Basics for New SAS Users",
SGF Proceedings, 2013, https://support.sas.com/resources/papers/proceedings13/120-2013.pdf

[Langston 2013] Rick Langston, "Submitting SAS Code On The Side",
SAS Global Forum 2013 Proceedings, 032-2013, https://support.sas.com/resources/papers/proceedings13/032-2013.pdf

[Werner 2014] Nina L. Werner, "Understanding Double Ampersand [&&] SAS Macro Variables",
MWSUG Proceedings, 2014, https://www.lexjansen.com/mwsug/2014/BI/MWSUG-2014-BI03.pdf

[Nayak 2015] Amar Nayak, "The Ampersand (&) Challenge, Single, Double or more?",
PHUSE Proceedings, 2015, https://www.lexjansen.com/phuse/2015/cc/CC08.pdf

[Wong & Short 2016] Kalina Wong & Sarah A. Short, "An Array of Fun: Macro Variable Arrays",
WUSS Proceedings, 2016, https://www.lexjansen.com/wuss/2016/88_Final_Paper_PDF.pdf

[Carpenter 2016] Arthur L. Carpenter, "Carpenter's Complete Guide to the SAS Macro Language, Third Edition",
SAS Institute Press, 2016

[Carpenter 2017] Arthur L. Carpenter, "Five Ways to Create Macro Variables: A Short Introduction to the Macro Language",
SGF Proceedings, 2017, https://support.sas.com/resources/papers/proceedings17/1516-2017.pdf

[Renauldo 2018] Veronica Renauldo, "Efficiency Programming with Macro Variable Arrays",
MWSUG Proceedings, 2018, https://www.lexjansen.com/mwsug/2018/SP/MWSUG-2018-SP-62.pdf

[Horstman 2019] Joshua M. Horstman, "Using Macro Variable Lists to Create Dynamic Data-Driven Programs",
MWSUG Proceedings, 2019, https://www.lexjansen.com/mwsug/2019/SP/MWSUG-2019-SP-053.pdf

[Huang 2020] Siqi Huang, "One Macro to create more flexible Macro Arrays and simplify coding",
PharmaSUG Proceedings, 2020, https://www.lexjansen.com/pharmasug/2020/AP/PharmaSUG-2020-AP-093.pdf

[McMullen 2020] Quentin McMullen, "A Close Look at How DOSUBL Handles Macro Variable Scope",
SAS Global Forum 2020 Proceedings, 4958-2020, https://support.sas.com/resources/papers/proceedings13/032-2013.pdf

[Jablonski 2020] Bartosz Jabłoński, "SAS Packages: The Way to Share (a How To)", SGF Proceedings, 2020, 4725-2020
https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2020/4725-2020.pdf
extended version available at: https://github.com/yabwon/SAS_PACKAGES/blob/main/SPF/Documentation

[Wang 2021] Mindy Wang, "Writing Out Your SAS Program Without Manually Writing it Out- Let Macros Do the Busy Work",
SGF Proceedings, 2021, https://communities.sas.com/t5/SAS-Global-Forum-Proceedings/
Writing-Out-Your-SAS-Program-Without-Manually-Writing-it-Out-Let/ta-p/726365

[Jablonski 2021] Bartosz Jabłoński, "My First SAS Package - a How To", SGF Proceedings, 2021 , 1079-2021
https://communities.sas.com/kntur85557/attachments/kntur85557/proceedings-2021/59/1/Paper_1079-2021.pdf
also available at: https://github.com/yabwon/SAS_PACKAGES/tree/main/SPF/Documentation/Paper_1079-2021
[Walker 2022] Andrew E. Walker, "Why Write Base SAS Code When the Macro Processor Can Do It for You",
SESUG Proceedings, 2022, https://www.lexjansen.com/sesug/2022/SESUG2022_Paper_213_Final_PDF.pdf
[Roudneva 2023] Ekaterina Roudneva, "Automating Reports Using Macros and Macro Variables",
WUSS Proceedings, 2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-172.pdf
[Jablonski(1) 2023] Bartosz Jabłoński, "A SAS Code Hidden in Plain Sight",
WUSS 2023 Proceedings, 189-2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-189.pdf
[Jablonski(2) 2023] Bartosz Jabłoński, "Share your code with SAS Packages a Hands-on-Workshop",
WUSS 2023 Proceedings, 208-2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-208.pdf

ACKNOWLEDGMENTS

The author would like to acknowledge Ted Clay and David Katz, they work was the inspiration.
The author would like to acknowledge Richann Watson (DataRich Consulting) and Quentin McMullen, the proofreading and linguistic contribution made this paper look and feel as it should!

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at one of the following e-mail addresses:

yabwon✉gmail.com or bartosz.jablonski✉pw.edu.pl

or via the following LinkedIn profile: www.linkedin.com/in/yabwon or at the communities.sas.com by mentioning @yabwon.

Appendix A - code coloring guide

The best experience for reading this article is in color and the following convention is used:

- The code snippets use the following coloring convention:

code: is surrounded by a black frame

```
In general we use black ink for the code but:
- code of interest is in red ink so that it can be highlighted,
- and comments pertaining to code are in a bluish ink for easier reading.
```

- The LOG uses the following coloring convention:

the log - is surrounded by a blueish frame

```
The source code and general log text are blueish.
Log NOTEs are green.
Log WARNINGSs are violet.
Log ERRORSs are red.
Log text generated by the user is purple.
```

Appendix B - install the SAS Packages Framework and the macroArray package

To install the SAS Packages Framework and a SAS Package we execute the following steps:

- First we create a directory to install SPF and Packages, for example: /home/user/packages or C:/packages.
- Next, depending if the SAS session has access to the internet:
  - if it does - we run the following code:

code: install from the internet

```
filename packages "/home/user/packages";

filename SPFinit url
 "https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas";
%include SPFinit;

%installPackage(SPFinit)
%installPackage(macroArray)
```

  - If the SAS session does not have access to the internet we go to the framework repository:

    https://github.com/yabwon/SAS_PACKAGES

    next (if not already) we click the stargazer button [★] ;-) and then we navigate to the SPF directory and we copy the SPFinit.sas file into the directory from step one (direct link: https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas). And for packages - we just copy the package zip file into the directory from step one.
- From now on, in all subsequent SAS session, it is enough to just run:

code: enable framework and load packages

```
filename packages "/home/user/packages";
%include packages(SPFinit.sas);
%loadPackage(macroArray)
```

to enable the framework and load packages. To update the framework or a package to the latest version we simply run:

code: update from the internet

```
%installPackage(SPFinit macroArray)
```

Appendix C - safety considerations

The SPF installation process, in a "nutshell", reduces to copying the `SPFinit.sas` file into the packages directory. It is the same for a packages too.

You may ask: *is it safe to install*?

Yes, it's safe! When you install the SAS Packages Framework, and later when you install packages, the files are simply copied into the packages directory that you configured above. There are no changes made to your SAS configuration files, or autoexec, or registry, or anything else that could somehow "break SAS." As you saw, you can perform a manual installation simply by copying the files yourself. Furthermore the SAS Packages Framework is:

- written in 100% SAS code, it does not require any additional external software to work,
- full open source (and MIT licensed), so every macro can be inspected.

When we work with a package, before we even start thinking about loading content of one into the SAS session, both the help information and the source code preview are available.

To *read help information* (printed in the log) you simply run:

code: get help info

```
%helpPackage(<packageName>, <*|componentName|license>)
```

To **preview source code of package components** (also printed in the log) you simply run:

code: get code preview

```
%previewPackage(<packageName>, <*|componentName>)
```

The asterisk means "print everything", the `componentName` is the name of a macro, or a function, or a format, etc. you want see.