

[DRAFT 2019.10.18] Paper 4725-2020:
SAS® packages - the way to share
(a how to)

Bartosz Jabłoński
Warsaw University of Technology / Citibank Europe PLC Poland

ABSTRACT

When working on Base SAS® code, especially when it becomes complex, there is a point in time when a developer decides to break it into small pieces. The developer creates separate files for macros, formats/informats, and for functions or data too. Eventually the code is ready and tested and it is time for the deployment. The issue is that the code had been written on a local Windows machine and the deployment is on a remote Linux server. Folders and files have to be moved in the proper structure, code has to be run in the right order and not mixed up. Moreover it is not the developer who is deploying... small challenge, isn't it?

How nice it would be to have it all (i.e. the code and its structure) wrapped up in a single file - a portable SAS package - which could be copied and deployed with a one-liner like: `%loadPackage(MyPackage)?`

In this article an idea of how to create such a "SAS-package" in a fast and convenient way will be proposed/shared. We will discuss:

- a concept of how to build a package,
- the tools required to do so, and
- a "how to" of the process (i.e. generating packages, loading, and using them).

The intended readers for the following document are intermediate SAS users (i.e. with good knowledge of Base SAS and practice in macro programming, see [1]) who want to learn how to share their code with others.

INTRODUCTION and CONTEXT

In the world of programmers, software developers, and "computer people" the concept of a package is well known and common one. To give an evidence of this statement let us consider three very popular examples the Linux, the Python, and the R software, and as an endorsement the following quotes.

According to [5]:

In Linux distributions, a "package" refers to a compressed file archive containing all of the files that come with a particular application. [...] Most packages also contain installation instructions for the OS, as well as a list of any other packages that are dependencies (prerequisites required for installation).

Common types of Linux packages include .deb, .rpm, and .tgz. Since Linux packages do not usually contain the dependencies necessary to install them, many Linux distributions use package managers that automatically read dependencies files and download the packages needed before proceeding with the installation.

According to [4]:

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual modules can then be cobbled together like building blocks to create a larger application. Packages allow for a hierarchical structuring of the module [...].

According to [3]:

In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others. As of January 2015, there were over 6,000 packages available on the Comprehensive R Archive Network, or CRAN, the public clearing house for R packages. This huge *variety of packages is one of the reasons that R is so successful*: the chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package. [As of October 2019, there were over 15,000 packages available on the Comprehensive R Archive Network!]

WHAT IS a SAS PACKAGE?

A **SAS package**¹ is an automatically generated, single, stand alone zip file containing organised and ordered code structures, created by the developer and extended with additional automatically generated "driving" files (i.e. descriptor, metadata, load, unload, and help files).

The purpose of a package is to be a simple, and easy in access, code sharing medium, which will allow: on the one hand, to separate the code's complex dependencies created by the developer from the user experience with the final product and, on the other hand, reduce developer's and user's unnecessary frustration related to a remote deployment process.

To create a package the developer must prepare the code files and a description file, fit them into a structured form (see the next section for details) and execute the %generatePackage() macro.

To use a package the user should download package's zip file into the packages' folder (containing the loadpackage.sas file). And, in the SAS session, the user should run the following code:

```
filename packages "<directory/containing/packages>";  
%include packages(loadpackage.sas);  
%loadPackage(packageName)
```

In two subsequent sections we introduce the concept of a SAS package from both user and developer's point of view. What is worth to mention, and what is one of the biggest advantage of using SAS packages, is that work to be done on the user's side to use provided package is almost non.

The last section provides an example in which a package is created.

¹The idea presented in this article should not be confused with other occurrences of "package" concept which could be found in the SAS ecosystem, e.g. Proc DS2 packages, SAS/IML packages, or SAS Integration Technologies Publishing Framework packages.

THE USER: HOW TO and THE RULES

User's files and folders. Since the idea of a SAS package is to take off (from the user's shoulders) the burden of "necessity to know how it is all connected and dependent" there are only few simple steps to be done on the user's end. The user's part of work required to use a package starts with setting up some files and folders, but is very short and in practice only the last step is repeated more than once. The work goes as follows:

- Create a folder for your packages, e.g. under Windows OS family C:/SAS_PACKAGES or under Linux/UNIX OS family /home/<username>/SAS_PACKAGES.
- Copy the loadpackage.sas file into packages' folder.
- Copy zip file with the package into packages' folder.

User's session. When all files and folders are settled the user, to enjoy the package, runs a SAS session and executes the following steps:

- For loading the package:

```
filename packages "<directory/containing/packages/>";
%include packages(loadpackage.sas);
%loadPackage(packageName)
```

- To get help information about the package printed in the log:

- for general information about the package run:
`%helpPackage(packageName)`
- for all available information about the package run:
`%helpPackage(packageName,*)`
- for a particular element of the package, e.g. function or macro, run:
`%helpPackage(packageName, entry)`

where *entry* is a single word which is used for context search. "License" prints out license text.

- For removing (a.k.a. unloading) package's content:

```
%unloadPackage(packageName)
```

After loading a package for the first time it is a *good practice* to read the log to find out more about packages content and the list of loaded elements.

Caution! There is one important *restriction* regarding the SAS session! Words "package" and "packages" are restricted as a file reference for the FILENAME statement and the FILENAME() function. These words are file references used internally by the %loadPackage(), %helpPackage(), and %unloadPackage() macros. Using them may cause unexpected results and may jeopardise package stability!

User's "under the hood". The above steps are all that are necessary to use and work with a package. There are also some additional things happening in the background. This section explains them in more details.

- The %loadPackage() macro loads all components of the package as a primary job. Additionally the package's header is printed into the log. Whenever an element of the package is loaded appropriate note is printed into the log. If there were any requirements provided they will be tested at this point.
- The %helpPackage() macro prints out into the log help information attached to the package's content. When no second argument is provided only package's description is printed out. When the second argument is provided, if it is an asterisk ("*") all help's content is printed out (for datasets also proc contents is run), if it is a helpKeyword then content search is executed based on its value and only selected parts of help's content are printed out. If helpKeyword value is "License" then package's license is printed out.
- The %unloadPackage() macro cleans up the session. All objects created by package (except execs) are deleted. If clean files were provided their content is executed too.

THE DEVELOPER: HOW TO and THE RULES

Developer's files and folders. The developer's part of work to build a package starts with preparing a set of files and folders. This part goes as follows:

- Create a folder for your package a.k.a. package's folder (hint: name it the same name as your package's name).
- Create a descriptor file, named `description.sas`, and copy it into the package's folder. The file is mandatory, has simple structure, and it contains package's metadata and (short) description (compare [3]). The structure of the `description.sas` file could be seen in the Figure 1.

Figure 1. Package descriptor structure.

```

/* This is the description file for the package.          */
/* The colon (:) is a field separator and is restricted */
/* in lines of the header part.                         */

/* **HEADER** */ ①
Type: Package : ②
Package: XXXXXXXX : ③
Title: Xxx xxxx xx XXXXXXXXX XXX : ④
Version: x.x : ⑤
Author: Fname1 Lname1 (xxx1@yyy.zz), Fname2 Lname2 (xxx2@yyy.zz) : ⑥
Maintainer: Fname3 Lname3 (xxx3@yyy.zz) : ⑦
License: XXX : ⑧
Encoding: XXXX : ⑨

Required: "Base SAS Software", "SAS/Xxx", "SAS/ACCESS Interface to Yyyy" : ①

/* **DESCRIPTION** */ ⑩
/* All the text below will be used in help */
DESCRIPTION START:

XXXXXXXXXXXX XXXXXXXX XXXXXX XXXXXXXX XXXXXXXX. XXXXXXXX
XXXX XXXXXXXXXXXXXX XX XXXXXXXXXXXXXX XXXXXXX. XXXXXXXX XXX
XXXX XXXXXX. XXXXXXXXXXXXXX XXXXXXXXXXXXXX XXXXXXXX.

DESCRIPTION END:

```

The meaning of entries (a.k.a. tags) inside the `description.sas` file content are the following (the dark bullet marks an element which is *mandatory*):

- (①) marks the header's start. Mind the structure of the tag, i.e. slash, asterisk, *space*, and double asterisk (`/* **`). Each of the following lines is a `key:value` pair and such pair must be a *single line of text*. The colon (:) is a field separator and is restricted in lines of the header.
- (②) is a constant (i.e. "Package"), required, and not null value for the key type.
- (③) is a package's name. It is required, not null, up to 24 characters long, and shares naming restrictions like those for a SAS dataset's name.
- (④) is short title of a package (i.e. one phrase). It is required and not null.
- (⑤) is a package's version, it is required and not null. (preferred form is: an integer value for a stable version, a decimal value for non-stale one)
- (⑥) and (⑦) are comma separated lists of package's author(s) and maintainer(s). Elements of lists are of the form: "Firstname Lastname (email@address.com)".
- (⑧) is a license under which the package is distributed. It is required, not null, possible values could be: MIT, GPL2, BSD, PROPRIETARY, etc. The license text itself should be inserted into a `license.sas` file (see the further steps).

- (❾) is the information under which SAS sessions encoding package's files were created. It is required and not null. Possible values could be: UTF8, WLATIN1, LATIN2, etc. and the values should satisfy requirements for the `encoding=` option of the `filename` statement.
- (❿) is the last required part. It is the description portion of the package. It is a free text bounded between the "DESCRIPTION START:" and the "DESCRIPTION END:" tags. It could be multi-line. It should elaborate about the package and its components (e.g. macros, functions, datasets, etc.)
- (❾) is a quoted and comma separated list of licensed SAS products required for the SAS session under which package will be used. Possible values inserted into the list should be the same as these the `proc setinit` prints in the log, e.g. "Base SAS Software", "SAS/IML", "SAS/ACCESS Interface to Teradata". The "Required" tag is optional, when it is empty or not provided in the description the testing code is not generated. Though, it is recommended to add this one.

Based on the header's information, the following internal macrovariables are generated: `packageName`, `packageVersion`, `packageTitle`, `packageAuthor`, `packageMaintainer`, `packageEncoding`, `packageLicense`.

- Inside the package's folder create subfolders for the code files. A subfolder's name must be structured as follows:
 - a) it contains only lower case letters, digits, and underscore ("_")
 - b) it is composed of two parts separated by and underscore ("_"), i.e.
 - the first part is a series of digits (with leading zeros, e.g. 001, 002, . . . , 123, 124, . . .); its purpose is to keep execution's sequence (e.g. in case the code must be ordered to run properly);
 - the second part, called folder's *type*, indicates subfolder's content. The *type* must be one of the following:

- `libname` (for libraries assignments),
`macro` (for macros),
`function` (for `proc fcmp` functions),
`format` (for formats and informats),
`data` (for the code generating dataset),
`exec` (for so called "free code") or
`clean` (for the code cleaning up the session after `execs`).

An example of a package's subfolders structure could be found in the Figure 2. In case the order of code's execution is irrelevant the first part (i.e. digits and underscore) may be skipped.

In case when the order of code's execution is important, e.g. `format $efg.` must be defined before `function abc()`, two folders of *type* `format` and `function` with two different sequence of digits have to be created in a way that digits maintain execution's order, e.g. `017_format` for the code of the `format $efg.` and `042_function` for the code of the `function abc()`.

The list of types may be extended in the future if need be.

- Copy the code's files into package's subfolders in accordance with *types* and the following set of rules:
 - One-file-one-object, e.g. a code of one function must be one file. The only exception are formats/informats, in this case one file may contains all four formats/informats sharing the same name, e.g. numeric format `abc.`, character format `$abc.`, numeric informat `abc.`, and character informat `$abc.` all must be kept in one file.
 - An object name is a file name, i.e. a definition of a macro named `%abc()` must be contained in a file named `abc.sas`.
 - The definition of a function must be enclosed in the following template of the `FCMP` procedure:

```
proc fcmp
  inlib  = work.&packageName.fcmp
  outlib = work.&packageName.fcmp.package
  <... other options ...>
;
  <... function's or subroutine's body ...>
run;
quit;
```

The `inlib=` and `outlib=` options are, literally, set to: `"work.&packageName.fcmp"` and `"work.&packageName.fcmp.package"`.

- o The definition of a format/informat must be enclosed in the following template of the `FORMAT` procedure:

```
proc format
  lib = work.&packageName.format
  <... other options ...>
;
  <... numeric format's definition ...>
  <... character format's definition ...>
  <... numeric informat's definition ...>
  <... character informat's definition ...>
run;
```

The `lib=` option is, literally, set to: `"work.&packageName.format"`.

- o `exec` folders are for so-called "free code", i.e. if a package, to be ready and usable, requires some additional code to be run (code not fitting provided *types*) - this code must be inserted into a file inside one of the `exec` subfolder.
- o `clean` folders are for cleaning after `execs`, i.e. if a code from one of `exec` folders creates some object (e.g. a catalog, a macro, or a dataset) the appropriate code inside a `clean` subfolder must be developed to remove created object.
- o Parts of code files which are to be used to generate help information must be enclosed between following text tags: `"/*** HELP START ***/"` and `"/*** HELP END ***/"`, e.g. from the following file containing: an example of a macro code, a help text, and other comments:

```

/*** HELP START ***
/* >>> %ABC() macro: <<<
*
* Main macro which allows to do
* this and that...
* Recommended for SAS 9.4 and higher.
*/
/*** HELP END ***

/* macro definition */
/*** HELP START ***
%MACRO ABC(
    param1 /* parameter 1 is used for ... */
    ,param2 /* parameter 2 is used for ... */
);
/*** HELP END ***
<.... body of a macro ...>
<...           ...>
<.... body of a macro ...>
%MEND ABC;

/*** HELP START ***
/* EXAMPLE 1: use in datastep

    data class;
        set sashelp.class;
        %ABC(age, weight)
    run;

*/
/*** HELP END ***

```

only the following parts of text will be extracted for help's purpose:

```

/* >>> %ABC() macro: <<<
*
* Main macro which allows to do
* this and that...
* Recommended for SAS 9.4 and higher.
*
*/
%MACRO ABC(
    param1 /* parameter 1 is used for ... */
    ,param2 /* parameter 2 is used for ... */
);
/* EXAMPLE 1: use in datastep

    data class;
        set sashelp.class;
        %ABC(age, weight)
    run;

*/

```

- Create a `license.sas` file containing license information for the package. Place the file in the package's folder (together with the `description.sas` and subfolders). If no file is provided the `license.sas` will be generated with standard MIT license (read "Generating package in practice - a use-case" section to see the MIT license text).

- Create a folder for packages, e.g. under Windows OS family C:/SAS_PACKAGES or under Linux/UNIX OS family /home/<username>/SAS_PACKAGES and copy the generatepackage.sas file into this folder.

Developer's session. When all files and folders are settled the developer runs SAS session and executes the following code:

```
filename packages "<directory/contsining/packages/>";
%include packages(generatepackage.sas);
/*ods html;*/
%generatePackage(filesLocation=<directory/with/package's/files/>)
```

When the %generatePackage macro ends its execution the packagename.zip file, containing all package's content inside it, is created inside the "<directory/with/package's/files/>".

Developer's "under the hood". Before reading this subsection further we recommend (for a better view) to have subsections "User's files and folders" and "User's session" of the "THE USER: HOW TO and THE RULES" section read.

When the packagename.zip file is created, by the %generatePackge() macro, a lot of things is happening behind the scenes. This section explains them in more details.

The first information the developer receives after process ends is a summary report displaying basic information about the package's content. In this summary the following elements are displayed: the package's location (i.e. folder), developer's &sysuserid., creation's timestamp, SAS version, the package's encoding information (based on the description.sas file), and current SAS session's encoding. From the description.sas file the package's name, version, and license type are extracted and printed. The last part of the summary is a table displaying a list of files used to build up the package.

Within the package's zip file we will find:

- Copies of all files from package's subfolders but with modified names, what is needed to keep the ordering in place. Each code file's name is extended with a prefix of a form: underscore, subfolder's name, and dot. For example if file name is abc.sas and subfolder's name is 007_macro then the new name is _007_macro.abc.sas.
- The description.sas file (the one described earlier) and the license.sas file.
- The packagemetadata.sas file containing definitions of internal macrovariables used by the %loadPackage(), %helpPackage(), and %unloadPackage() macro.
- The load.sas file containing code executed by the %loadPackage() macro. The file's content is built based on the subfolders and files' structure provided by the developer. The file is a series of %includes, with additional automatic note comments in %put statements, and, if need be, set of options modifications e.g. inserts to fmtsearch option for formats/informats or appends to cmplib option for functions. If files of type exec are inside the package a code printing out their content into the log is also attached.
- The help.sas file containing code executed by the %helpPackage() macro. The file contains 1) code which displays general package's description, 2) code which search for a content based on helpKeyword's value and prints out the information, and 3) code which, if helpKeyword's value is "License", prints out the license text.
- The unload.sas file containing code executed by the %unloadPackage() macro. The file's content is built based on the subfolders and files structure provided by the developer. Code inside the file removes macros, functions, formats, datasets and libraries created during loading process. It restores fmtsearch and cmplib options. If clean type subfolder is provided files from within the folder are %included (they are executed as the first).

Figure 2. Example of a package's subfolders structure.

```

<packageName>
.
.
.
+-000_libname [one file one libname]
|       |
|       +-abc.sas [a file with a code creating libname ABC]
|
+-001_macro [one file one macro]
|       |
|       +-hij.sas [a file with a code creating macro HIJ]
|       |
|       +-klm.sas [a file with a code creating macro KLM]
|
+-002_function [one file one function,
|       |   option OUTLIB= should be: work.&packageName.fcmp.package
|       |   option INLIB=  should be: work.&packageName.fcmp
|       |   (both literally with macrovariable name and "fcmp" suffix)
|       |
|       +-efg.sas [a file with a code creating function EFG]
|
+-003_format [one file one format,
|       |   option LIB= should be: work.&packageName.format
|       |   (literally with macrovariable name and "format" suffix)
|       |
|       +-efg.sas [a file with a code creating format EFG and informat EFG]
|
+-004_data [one file one dataset]
|       |
|       +-abc.efg.sas [a file with a code creating dataset EFG in library ABC]
|
+-005_exec [so called "free code", content of the files will be printed
|       |   to the log before execution]
|       |
|       +-<no file, in this case folder may be skipped>
|
+-006_format [if your codes depend eachother you can order them in folders,
|       |   e.g. code from 003.... will be executed before 006....]
|       |
|       +-abc.sas [a file with a code creating format ABC,
|                   using the definition of the format EFG]
+-007_function
|       |
|       +-<no file, in this case folder may be skipped>
|
+-<sequential number>_<type [in lower case]>
|
+-...
|
+-00n_clean [if you need to clean something up after exec file execution,
|       |   content of the files will be printed to the log before execution]
|       |
|       +-<no file, in this case folder may be skipped>
+-...
|
...

```

Generating package in practice - a use-case

The practical **example** will be build based on one of author's favorite SAS article, namely Mike Rhoads' *"Use the Full Power of SAS in Your Function-Style Macros"* [2], which introduces the macro-function-sandwich programming approach. The idea is to allow user to execute SQL's "select" code within a datastep, e.g.

```
data class_subset;
  set %SQL(select
            name
            , sex
            , height
            from
            sashelp.class
            where
            age > 12
          )
  (rename=(height=heightInch));

  heightCm = 2.54 * heightInch;
run;
```

Thus the package's name will be *SQLinDS* and it will be providing the `%SQL()` macro which allow users to write queries like the one above. Internally the `%SQL()` macro uses a user defined function, another macro, and stores intermediate data (views) inside a predefined library (pointing to a subdirectory of the `work`). Package will be build with 5 files: `description.sas`, two `macros`, one `function`, and one `library`. Let's assume we created the following package's folder `C:/SAS_PACKAGES/SQLinDS/` and we copied `generatepackage.sas` file into the `C:/SAS_PACKAGES/` directory. The structure of subfolders created for the package is presented in the Figure 3.

Figure 3. *SQLinDS* package - subfolders' structure.

```
<C:/SAS_PACKAGES/SQLinDS/>
  .
  |
  +-description.sas ①
  |
  +-000_libname
  |
  |       |
  |       +-dssql.sas ②
  |
  +-001_macro
  |
  |       |
  |       +-dssql_inner.sas ③
  |
  |       |
  |       +-sql.sas ⑤
  |
  +-002_function
  |
  |       |
  |       +-dssql.sas ④
  |
  +-license.sas ⑥
```

The content of files composing the package is following:

1 /* This is the description file for the package. */
/* The colon (:) is a field separator and is restricted */
/* in lines of the header part. */

/* **HEADER** */
Type: Package
Package: SQLinDS
Title: SQL queries in Data Step
Version: 1.0
Author: Mike Rhoads (RhoadsM1@Westat.com)
Maintainer: Bartosz Jablonski (yabwon@gmail.com)
License: MIT
Encoding: UTF8

/* **DESCRIPTION** */
/* All the text below will be used in help */
DESCRIPTION START:

The SQLinDS package is an implementation of
the macro-function-sandwich concept introduced in:
"Use the Full Power of SAS in Your Function-Style Macros"
the article by Mike Rhoads, Westat, Rockville, MD

Copy of the article can be found at:
<https://support.sas.com/resources/papers/proceedings12/004-2012.pdf>

SQLinDS package provides following components:
1) %dsSQL_inner() macro
2) dsSQL() function
3) %SQL() macro

Library DSSQL is created in a subdirectory of the WORK library.
DESCRIPTION END:

②

```
/**> HELP START */

/* >>> dsSQL library: <<<
*
* The dsSQL library stores temporary views
* generated during %SQL() macro's execution.
* If possible, created as a subdirectory of WORK:
*
options dlCreateDir;
LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))/dsSQLtmp";
*
* if not then redirected to WORK
*
LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))";
*/
/**> HELP END */

data WORK._%sysfunc(datetime(), hex16.)_;
length option $ 64;
option = getoptoption("dlCreateDir");
run;

options dlCreateDir;

data _null_;
set _LAST_;
rc1 = LIBNAME("dsSQL", "%sysfunc(pathname(work))/dsSQLtmp", "BASE");
rc2 = LIBREF("dsSQL");
if rc2 NE 0 then
  rc1 = LIBNAME("dsSQL", "%sysfunc(pathname(work))", "BASE");
call execute ("options " || strip(option) || ";");
run;

proc delete data = _last_;
run;

libname dsSQL LIST;
```

③

```
/**> HELP START **/
```

```
/* >>> %dsSQL_Inner() macro: <<<
*
* Internal macro called by dsSQL() function.
*
* Recommended for SAS 9.3 and higher.
* Based on paper:
* "Use the Full Power of SAS in Your Function-Style Macros"
* by Mike Rhoads, Westat, Rockville, MD
* https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
*
**/
```

```
/**> HELP END **/
```

```
/* inner macro */
%MACRO dsSQL_Inner() / SECURE;
  %local query;
  %let query = %superq(query_arg);
  %let query = %sysfunc(dequote(&query));

  %let viewname = dsSQL.dsSQLtmpview&UNIQUE_INDEX_2.;
  proc sql;
    create view &viewname as
      &query
    ;
  quit;
%MEND dsSQL_Inner;
```

4

```
/* *** HELP START ***/


/* >>> dsSQL() function: <<<
*
* Internal function called by %SQL() macro.
*
* Recommended for SAS 9.3 and higher.
* Based on paper:
* "Use the Full Power of SAS in Your Function-Style Macros"
* by Mike Rhoads, Westat, Rockville, MD
* https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
*
**/


/* *** HELP END ***/



proc fcmp
  inlib  = work.&packageName.fcmp
  outlib = work.&packageName.fcmp.package
;
function dsSQL(unique_index_2, query $) $ 41;
  length
    query query_arg $ 32000 /* max querry length */
    viewname $ 41
  ;
  query_arg = dequote(query);
  rc = run_macro('dsSQL_Inner' /* <-- inner macro */
                 ,unique_index_2
                 ,query_arg
                 ,viewname
                 );
  if rc = 0 then return(trim(viewname));
  else
    do;
      put 'ERROR:[function dsSQL] A problem with the dsSQL() function';
      return(" ");
    end;
  endsub;
run;
quit;
```

5

```

/*** HELP START ***/


/* >>> %SQL() macro: <<<
*
* Main macro which allows to use
* SQL's queries in the data step.
* Recommended for SAS 9.3 and higher.
* Based on paper:
* "Use the Full Power of SAS in Your Function-Style Macros"
* by Mike Rhoads, Westat, Rockville, MD
* https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
*
* EXAMPLE 1: simple sql querry

data class_subset;
  set %SQL(select name, sex, height from sashelp.class where age > 12);
run;

* EXAMPLE 2: with dataset options

data renamed;
  set %SQL(select * from sashelp.class where sex = "F")(rename = (age=age2));
run;

* EXAMPLE 3: dictionaries in datastep

data dictionary;
  set %SQL(select * from dictionary.macros);
run;

**/


/*** HELP END ***/


/* outer macro */
%MACRO SQL() / PARMBUFF SECURE;
  %let SYSPBUFF = %superq(SYSPBUFF); /* macroquoting */
  %let SYSPBUFF = %substr(&SYSPBUFF, 2, %LENGTH(&SYSPBUFF) - 2); /* remove brackets */
  %let SYSPBUFF = %superq(SYSPBUFF); /* macroquoting */
  %let SYSPBUFF = %sysfunc(quote(&SYSPBUFF)); /* quotes */
  %put NOTE-***the querry***; /* print out the querry in the log */
  %put NOTE-&SYSPBUFF. ;
  %put NOTE-*****;

  %local UNIQUE_INDEX; /* internal variable, a unique index for views */
  %let UNIQUE_INDEX = &SYSINDEX;
  %sysfunc(dsSQL(&UNIQUE_INDEX, &SYSPBUFF)) /* <-- call dsSQL() function,
                                             see the WORK.SQLInDSfcmp dataset */

%MEND SQL;

```

⑥ MIT license text (used by default, mind macrocode in the first line):

```
Copyright (c) %sysfunc(today(),year4.) &packageAuthor.
```

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

When all files are placed inside proper subfolders we start a new SAS session and we execute the following code:

```
filename packages "C:/SAS_PACKAGES/";
%include packages(generatepackage.sas);
/*ods html;*/
%generatePackge(filesLocation=C:/SAS_PACKAGES/SQLinDS/)
```

As a result, extra the summary report, we receive (inside the C:/SAS_PACKAGES/SQLinDS/ folder) the sqlinds.zip file. Our package is prepared. And ready for sharing!

THE CODE

If you are interested in testing approaches presented above yourself and want to play a bit with the code and data you can download SAS codes which were the motivation for this paper under the following "world wild web" address:

http://www.mini.pw.edu.pl/~bjablons/SASpublic/SAS_PACKAGES

or from authors GitHub:

https://github.com/yabwon/SAS_PACKAGES

REFERENCES

- [1] Art Carpenter, "Carpenter's Guide to Innovative SAS Techniques", SAS Press
- [2] Mike Rhoads, "Use the Full Power of SAS in Your Function-Style Macros",
SAS Global Forum 2012 Proceedings, <https://support.sas.com/resources/papers/proceedings12/004-2012.pdf>
- [3] Hadley Wickham, "R Packages: Organize, Test, Document, and Share Your Code",
O'Reilly Media 2015, <http://r-pkgs.had.co.nz/description.html>
- [4] <https://realpython.com/python-modules-packages/> as of October 2019
- [5] <https://www.internetblog.org.uk/post/1520/what-is-a-linux-package/> as of October 2019

CONTACT INFORMATION

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

yabwon@gmail.com

bartosz1.jablonski@citi.com

or via the following LinkedIn profile:

www.linkedin.com/in/yabwon

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.