SAS® GLOBAL FORUM 2020 #SASGF

Paper 4725-2020 ◄ Extended Version, 2024.10.27 ►

SAS® Packages - The Way To Share (a How To)

Bartosz Jabłoński, Warsaw University of Technology

ABSTRACT

When working on Base SAS® code, especially when it becomes more and more 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[1] and files have to be created with 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 (framework), 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. To give an evidence of this statement let us consider four very popular examples: Linux, Python, TEX, and R software, and as an endorsement the following quotes.

According to [**6**]:

> "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 [**5**]:

> "[In Python] 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 [...]."

[1] Folders - also known as Directories

According to [**4**]:

> "Many LaTeX commands [...] are not specific to a single class but can be used with several classes. A collection of such commands is called a package and you inform LaTeX about your use of certain packages in the document by placing one or more `\usepackage` commands after `\documentclass`.
>
> Just like the `\documentclass` declaration, `\usepackage` has a mandatory argument consisting of the name of the package and an optional argument that can contain a list of package options that modify the behaviour of the package."

As of January 2020, there were over 5,700 packages available on the Comprehensive TeX Archive Network! More than a dozen of TeX packages were used while writing this article.

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 January 2020, there were over 15,000 packages available on the CRAN!

The SAS/IML offers (limited) version of functionality similar to concept of a package in programming languages and environments mentioned above but such functionality is not available in the Base SAS. The main goal of this article is to propose, describe, explain, and discuss an original idea of a process and tools required to build SAS packages. In subsequent sections we introduce the concept of a SAS package from both user and developer point of view. What is worth to mention, and what is one of the biggest advantages of using SAS packages, is that the work to be done on the user's side to use a package is almost none. The last section provides an example of package creation.

WHAT IS a SAS PACKAGE?

A **SAS package** [2] 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. description, metadata, load, unload, and help files).

The purpose of a package is to be a simple, and easy to access, code sharing medium, which will allow: on the one hand, to separate the code 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.

In this article we are presenting a *standalone Base SAS framework* which allows to develop and use SAS packages.

To create a package the developer must prepare the code files and a description file, fit them into a structured form (see next sections for details), download `%SPFinit.sas` (the SAS Packages Framework) file and execute the `%generatePackage()` macro (see "TUTORIAL" subsection).

To use a package the user should download the package `zip` file into the packages folder (containing also downloaded `SPFinit.sas` file). And, in the SAS session, the user should run the following code:

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

to have the package available.

[2] The 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, SAS ODS 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 a 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 a packages folder. 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`.

User's session. When folder is ready the user, to enjoy the package in a SAS session, executes the following steps.

- Assigns the `packages` filename to the packages folder:
  ```
  filename packages "<directory/containing/packages/>";
  ```
- There are two ways to enable the framework (i.e. set up the SAS session for using packages):
  - for a "one-time-only" set up (framework is enabled only for current SAS session) the user runs:
    ```
    filename SPFInit url
      "https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas";
    %include SPFInit; /* enables the framework */
    ```
  - for a "permanent" set up (the framework file is downloaded for use in the current and future SAS sessions) user: *downloads* the `SPFinit.sas` file into the packages folder (see "TUTORIAL" subsection for details) and, to enable framework for the SAS session (current or in future), runs:
    ```
    %include packages(SPFinit.sas);
    ```
- To install the package the user either *downloads the package `zip` file into the packages folder manually* (see "TUTORIAL" subsection for details) or executes the following code (assuming the framework is already enabled):
  ```
  %installPackage(packageName)
  ```
  to download the package automatically from the Internet repository.
- To get help about the package printed in the log:
  - for general information about the package the user runs:
    ```
    %helpPackage(packageName)
    ```
  - for all available information about the package the user runs:
    ```
    %helpPackage(packageName,*)
    ```
  - for a particular element of the package, e.g. a function or a macro, help the user runs:
    ```
    %helpPackage(packageName, helpKeyword)
    ```
    where *helpKeyword* is a single word which is used for context search. "License" prints out license text.
- To load the package into current SAS session the user runs:
  ```
  %loadPackage(packageName)
  ```
- For removing (a.k.a. unloading) the package content from the current SAS session the user runs:
  ```
  %unloadPackage(packageName)
  ```

For a detailed list of macro parameters either see the "Appendix B" or from the SAS session level run macros with *empty* parameter list, e.g. `%loadPackage()`.
After loading a package for the first time it is a *good practice* to read the log to find out more about package content and the list of loaded elements.
The `%loadPackage()` is provided with an extra macro wrapper named `%loadPackageS()` (mind the S at the end). The wrapper allows to load multiple packages with one run, list of packages for loading has to be comma separated, e.g. `%loadPackageS(muPackage, otherPackage, PiPackage)`.
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!

Note. The code executed behind the scenes is designed to affect the user session environment in an "as small as possible" way. The required minimum is compilation of the SAS Packages Framework macros (%generatePackage(), %installPackage(), %loadPackage(), %unloadPackage(), %helpPackage(), %listPackages(), %loadPackageS(), %verifyPackage(), %extendPackagesFileref(), and %loadPackageAddCnt()) and setting a global macrovariable SYSloadedPackages. When the %loadPackage() macro is executed for the first time the SYSloadedPackages macrovariable is created and its value is updated with package name and version for each new loaded package. If a package contains IML modules or CASL user defined functions additional macros are generated too, those macros are named %<packageName>IML() and %<packageName>CASLudf() respectively (see User's "under the hood").

Note. The packages filename can point to multiple directories, e.g.

```
filename packages ("/dir/nmbr/one" "/dir/nmbr/two");
```

The directories list is searched from left to right both for the framework and for packages. During installation the first directory is used. To extend an existing packages filename with a new directory the %extendPackagesFileref() utility-macro can be used, e.g. to add /new/dir directory the following code can be executed: filename packages ("/new/dir" %extendPackagesFileref());

Note. Because of a bug described in https://support.sas.com/kb/68/708.html it is *not recommended*, for SAS earlier than 9.4M7, to store packages in folders containing local characters, e.g.

filename packages "/dir/ŻÓŁĆ/pckg"; or filename packages "C:\folder\空手\pckg";

The directories with such local characters cause error in work of the framework and packages.

Package additional content. Usually a package is a "code container" with plain text files with code inside, but sometimes a package may also contain so called "*additional content*". Additional content is provided by a Developer who want to include, for example: a 150 pages long pdf file with detailed documentation, or a bunch of jpg files with figures depicting relations between package macros, or something else. Basically content that can't be delivered in the "standard" way. By *default* this additional content is not deployed automatically e.g., from security point of view, or production job doesn't need it to run, etc. But if there is a need for it there are three ways to get it:

The first one ("*by-the-book*"), and also the recommended one. The additional content is extracted during the automatic installation process using the %installPackage() macro. For this to work the loadAddCnt= parameter has to be set to 1. The additional content is extracted to the <packageName>_AdditionalContent directory into the same location where the package is installed i.e., inside packages fileref location.

The second one ("*by-the-work*"), when the additional content is extracted during the loading process with the %loadPackage() macro. For this to work also the loadAddCnt= parameter has to be set to 1. The additional content is extracted to the <packageName>_AdditionalContent directory inside the Work library location.

The third one ("*by-the-user*"), when the additional content is extracted with dedicated %loadPackageAddCnt() macro. By default the additional content is extracted to the <packageName>_AdditionalContent directory inside the Work library location too, but it can be altered by changing the target= parameter, which indicates the location.

If done "*by-the-book*", or "*by-the-user*" with target= parameter, the additional content is not automatically deleted when SAS session ends, in this case the "additionals" have to be deleted manually by the User.

To verify if a package has an additional content available read the log after running the %helpPackage() macro.

User's "under the hood". The above steps are all that is 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 %installPackage() macro installs a package, i.e. it downloads the package zip file from the Internet repository into the folder pointed by the packages filename. Successful installation ends with return code 0 in the log. By default if the package is already installed the macro will overwrite existing package file. A space separated list of packages may be provided to install multiple packages at once, e.g. %installPackage(somePackage PiPackage myPackage) If the SPFinit or SASPackagesFramework is used as the macro argument then the SAS Packages Framework file SPFinit.sas is downloaded into the packages folder. Current default packages repository address is: https://raw.githubusercontent.com/SASPAC/ but it could be altered with the sourcePath= parameter. Also the mirror= parameter allows to alter source for repository.
  Value mirror=0 indicates: https://raw.github.com/SASPAC/
  Value mirror=1 indicates: https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main
  value mirror=2 indicates: https://pages.mini.pw.edu.pl/~jablonskib/SASpublic/SAS_PACKAGES
  The default value is 0.
  The version parameter indicates which historical version of a package to install. Historical version are available only if mirror parameter is set to 0. Default value is null which means "install the latest". Starting October 2022 the %installPackages() macro allows to install multiple packages with version requirement. The functionality works for mirror=0 parameter, for example: %installPackage(SPFinit(20201002) macroarray(1.0) dfa(0.5) GSM). For mirror 1 and 2 a note about lack of version support is printed in the log. Starting September 2024 the %installPackages() macro allows to create a macrovariable containing information about successful and failed installations. Name of the macrovariable is provided with SFRCVN parameter, a global macrovariable is created. The format of returned value is <number of successes>.<number of failures>, e.g. value 4.2 means four successes and 2 failures out of 6 requests.

- The %loadPackage() macro loads all components of the package as a primary job. Additionally information from the package description file is printed into the log. Whenever an element of the package is loaded an appropriate note is printed into the log. If a macro is loaded and a macro with the same name is found in the sasmacr catalog a note about this fact is printed. If there were any requirements provided they will be tested at this point. If required element of the SAS system is missing the loading process is aborted with an error message. If required package is not loaded (SYSloadedPackages macrovariable is tested) SAS tries to load that required package, in case of failure loading process is aborted with an error message. If the loading process is successful the SYSloadedPackages macrovariable is updated with new entry. If a particular version of a package is required then the %loadPackage() macro has to be executed with additional parameter requiredVersion, e.g. %loadPackage(PiPackage, requiredVersion=3.1415). The macro tests if the provided version is greater o equal then the required version, in case of failure loading is aborted with an error message (this condition *implicite* assumes that packages are backward compatible).
  Note. The package may contain datasets as so called "*lazy loading datasets*". Such datasets are not automatically loaded when the %loadPackage() macro is executed. To load such dataset the user has to call the %loadPackage() macro with non missing lazyData= parameter, e.g %loadPackage(PiPackage, lazyData=work.first1E6digits) (list of multiple elements separated by space is allowed, an asterisk(*) means "load all data"). In case when regular dataset provided during loading process has to be reloaded the lazyData= parameter can be used to do that.
  Note. Sometimes a package offers so many features that the number may be "overwhelming". In such case only some of them may be selected for loading. Such process is called a "*cherry picking*". The feature is provided by the %loadPackage() macro which uses a cherryPick= parameter.
  For example, execution of the following code:

`%loadPackage(BasePlus, cherryPick=rainCloudPlot getVars)`
results with loading only the `rainCloudPlot` and the `getVars` elements. If several object types (e.g., a macro and a format) share the same name all will be loaded.
The expected value of the `cherryPick=` parameter is a *space separated* list of selected elements of the package to be loaded into the SAS session. Default value of an asterisk (`*`) means: "load all elements of the package". Also the empty list is equivalent to default.
What is the trade-off?
  - Since the cherry picking selects only a part of the package the `SYSloadedPackages` macrovariable is not updated with the package name (unless the package was already loaded).
  - Dependencies i.e., packages from the `ReqPackages` list, are not loaded automatically, so they have to be loaded manually.
  - The `%unloadPackage()` macro executed on such partially loaded package may sometimes issue some irrelevant warnings e.g., "Subroutine 'XYZ' cannot be found for deletion".

  If a package contains `exec` type files they loading can be suppressed by setting the `suppressExec=` parameter to one.

- The `%helpPackage()` macro prints out into the log help information attached to the package content. The `%helpPackage()` macro is independent from the `%loadPackage()` macro what means that it can be executed even if the `%loadPackage()` was not executed. The user can read about a package before loading it! When no second argument is provided only the package description, list of package elements, list of required components, and version of the `%generatePackage()` macro used are printed out. When the second argument is provided, if it is an asterisk ("`*`") all help 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 content are printed out. If `helpKeyword` value is "License" then package license is printed out. List of components of a package can be saved into a dataset named `work.packageName_content` by setting paramater `packageContentDS=` to one.

- The `%unloadPackage()` macro cleans up the session. All objects created by the package (except `execs`) are deleted. If `clean` files were provided their content is executed too. If the unloading process is successful the `SYSloadedPackages` macrovariable is updated. If a package required additional packages to be loaded then during unload a list of calls to `%unloadPackage()` for this additional packages is printed in the log, to allow their manual execution.
- The `%listPackages()` macro prints in the log list of SAS packages availabe in the `packages` fileref.
- The `%verifyPackages()` macro, for SAS 9.4M6 and later, allows to verify if given package has the same hash (SHA256) digest as the one provided. The Developer provides the User with hash digest of the zip file which is required to verify the package. The `hashing_file()` function is used to generate hash identifier. Example of use would be: `%verifyPackageS(myPackage, hash=F*VBKSH93VB39RBKJBVC0LBVJBU21RASDL80BBK221HMBIOP)`
- The `%previewPackage()` macro prints in the log the code of SAS package elements. Everything, including comments, is printed out so the code can be inspected without necessity of unzipping package files by hand. The macro behaviour is analogues to the behaviour of the `%helpPackage()` macro. The second argument is exactly the same *helpKeyword* which is used by the `%helpPackage()` macro (except for the *License* keyword which is ignored).

User's utility macros for loading package content. During execution of the `%loadPackage()` macro, if a package contains IML modules or CASL user defined functions additional utility macros for IML Modules and CASL UDFs are generated when package is loaded. Macros are generated with the following names: `%<packageName>IML()` and `%<packageName>CASLudf()`. Their purpose is to make loading of Modules or UDFs (with potentially multiple dependencies) easy in `Proc IML` and `Proc CAS`.

Run them, accordingly, as the first line in the `Proc IML` or `Proc CAS` to access the package content. For `Proc IML` the use is as follows:

```
proc IML;
  %<packageName>IML()
  <... your code using IML modules from the package ...>
quit;
```

For Proc CAS the use is as follows:

```
proc CAS;
  %<packageName>CASLudf()
  <... your code using CASL UDFs from the package ...>
quit;
```

If a utility macro is generated appropriate note and a code snippet is printed in the log of the package loading process. In 99% cases macros are used with default parameters values but, in case when deeper insight about macros parameters is needed, help info is printed in the log when the following code is run: `%<packageName>IML(list=HELP)` or `%<packageName>CASLudf(list=HELP)`
If created, those macros are automatically deleted when the `%unloadPackage()` macro is run.

User's "in case of emergency". This section covers "emergency" situations that user may come across when for the first time loads a package.
The first one is *backward compatibility*. The package file is a `zip` file so at least SAS 9.4 is required to use it. But it is also possible to use earlier versions of SAS software if it is needed. When the SAS session does not support ZIP fileref the following solution could be used: unzip the `packagename.zip` file content into a `packagename.disk` folder and run loading, helping, and unloading macros with following options:

```
%loadPackage(packageName, zip=disk, options=)
%helpPackage(packageName, helpKeyword, zip=disk, options=)
%unloadPackage(packageName, zip=disk, options=)
```

The second one is *no access to the `SPFinit.sas` framework file*. When user has only the `packagename.zip` file with the package but does not have an access to the `SPFinit.sas` file with the `%loadPackage()` macro the following solution could be used: each package generated by `%generatePackage()` macro contains `%ICEloadPackage()` macro, which is a simplified version of the `%loadPackage()` macro, and can be used for an "emergency" load. To use it: a) make file reference to the package zip file, b) include `iceloadpackage.sas`, c) make file reference to the package folder, and d) load package.

```
filename ice ZIP "<directory/containing/packages>/packagename.zip";
%include ice(iceloadpackage.sas);
filename packages "<directory/containing/packages>/";
%ICEloadPackage(packageName)
```

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 folder (hint: name it the same as your package).
- Create a description file, named `description.sas`, and copy it into the package folder. The file is mandatory, has simple structure, and it contains package metadata and (short) description (compare [**3**]). The simple structure of the `description.sas` file can be seen in Figure 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** */ ❶
Type: Package                                              : ❷
Package: ThePackageName                                    : ❸
Title: Short description, single sentence.                 : ❹
Version: x.y.z                                             : ❺
Author: Fname1 Lname1 (xxx1@yyy.zz), Fname2 Lname2 (xxx2@yyy.zz) : ❻
Maintainer: Fname3 Lname3 (xxx3@yyy.zz)                    : ❼
License: XYZ17                                             : ❽
Encoding: UTF8                                             : ❾

Required: "Base SAS Software", "SAS/Xxx", "SAS/ACCESS Interface to Yyyy" : ①
ReqPackages: "somePackage (3.14)", "otherPackage (42)"                   : ②

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

Lorem ipsum dolor sit amet, ThePackageName consec tetur
adipis cingelit. Nullamdapibus lacus a elit congue
elementum. Suspendisse iaculis ipsum nec ante luctus
volutpat. Donec iaculis laoreet tristique.

DESCRIPTION END:
```

Figure 1: Package description structure.

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

(❶) `/* **HEADER** */` - marks the header start (it is optional comment). Each of the following lines is a `key:value` pair and such a pair must be a *single line of text*. The colon (`:`) is a field separator and is restricted in lines of the header.

(❷) `Type` - is a constant (i.e. "`Package`"), required, and not null value.

(❸) `Package` - is the package name. It is required, not null, up to 24 characters long, and shares naming restrictions like those for a SAS dataset name.

(❹) `Title` - is the short title of a package (i.e. one phrase). It is required and not null.

(❺) `Version` - is the package version, it is required, not null, and a positive number (i.e. > 0). The preferred form is: an integer value for a stable version, a decimal value for a non-stable one. Since the `%loadPackage()` macro contains a dedicated macrovariable `requiredVersion` to tests if the provided version of a package is *greater or equal* then the required version - a very important, in line with the "SAS way", assumption must be highlighted! The assumption is: *packages are assumed to be backward compatible*. The `major.minor.patch` approach is also accepted. A package version can be: `X`, `X.Y`, or `X.Y.Z`. Missing parts are treated as 0, e.g. 1 is equivalent with 1.0.0, 1.2 is 1.2.0, and .2 is 0.2.0, etc.

(❻) and (❼) `Author, Maintainer` - are comma separated lists of package author(s) and maintainer(s). Elements of lists are of the form: "`Firstname Lastname (email@address.com)`".

(❽) `License` - is the license under which the package is distributed. It is required, not null, possible values are: MIT, GPL2, BSD, PROPRIETARY, etc. The license text itself should be inserted into the `license.sas` file (see further steps).

(❾) `Encoding` - is the information about SAS sessions encoding the package files were created in. It is required and not null. Possible values are: UTF8, WLATIN1, LATIN2, etc. and the values should satisfy requirements for the `encoding=` option of the `filename` statement.

(❿) `/* **DESCRIPTION** */` - Marks the description start (it is optional comment). The package description is the last required part. 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.) Text outside the tags is ignored.

(①) `Required` - is a quoted and comma separated list of licensed SAS products required for the SAS session under which the 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.

(②) `ReqPackages` - is a quoted and comma separated list of names and versions (in parentheses) of other SAS packages required for the package to work. Possible values inserted into the list should be formatted like e.g. `"SQLinDS (0.1)"`. The "`ReqPackages`" tag is optional, when it is empty or not provided in the description the testing code is not generated.

The `description.sas` file without additional comments could look like that:

```
Type: Package
Package: ThePackageName
Title: Short description, single sentence.
Version: x.y
Author: Fname1 Lname1 (xxx1@yyy.zz)
Maintainer: Fname2 Lname2 (xxx2@yyy.zz)
License: MIT
Encoding: UTF8
Required: "Base SAS Software", "SAS/IML"
ReqPackages: "somePackage (3.14)", "otherPackage (42)"

DESCRIPTION START::::::::::::::::::::::::::::::::::::

Lorem ipsum dolor sit amet, ThePackageName consec tetur
adipis cingelit. Nullamdapibus lacus a elit congue
elementum. Suspendisse iaculis ipsum nec ante luctus
volutpat. Donec iaculis laoreet tristique.

DESCRIPTION END::::::::::::::::::::::::::::::::::::
```

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

- Inside the package folder create subfolders for the code files. A subfolder name has to be structured as follows:
  a) it contains only lower case letters, digits, and underscore ("_")
  b) it is composed of two parts separated by an 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 sequence in case the code must be ordered to run properly;

- the second part, called folder *type*, indicates subfolder content. The *type* has to be one of the following:
  `libname` (for libraries assignments),
  `macro` (for macros),
  `function` (for `proc fcmp` functions),
  `functions` (mind the "s" at the end, for `proc fcmp` functions),
  `format` (for formats and informats),
  `formats` (mind the "s" at the end, for formats and informats),
  `imlmodule` (for IML modules),
  `casludf` (for CASL user defined functions),
  `proto` (for `proc proto` external C functions),
  `data` (for the code generating datasets),
  `lazydata` (for the code generating datasets which will be loaded on demand, so called "lazy loading datasets"),
  `exec` (for so called "free code"),
  `clean` (for the code cleaning up the session after `exec`s),
  `kmfsnip` (for the code of KMF-abbreviation snippets for DMS or EG sessions),
  `ds2pck` (for the code of PROC DS2 packages),
  `ds2thr` (for the code of PROC DS2 threads),
  `test` (for developer code with package tests) or
  `addcnt` (for additional content for the package).

  An example of a package subfolders structure can be found in Figure 2. In case the order of code execution is irrelevant the first part (i.e. digits and underscore) may be skipped.
  In case when the order of code execution is important, e.g. format `$efg.` must be defined before function `abc()`, two folders of *type* `format` and `function` with two different sequences of digits have to be created in a way that digits indicate execution order, e.g. `017_format` for the code of the format `$efg.` and `042_function` for the code of the function `abc()`.
  Note. The list of types may be extended in the future if need be.
  Note. If folder name starts with `!` (e.g., `!ignore_me`) or *type* of the folder is "unknown" (e.g., not supported one) the content of such folder is ignored during package generation process.
- Copy the files with the code into package subfolders in accordance with *types* and the following set of rules:
  - One-file-one-object, e.g. macro `%abc()` definition has to be contained in a single file without any definitions of other objects. The only exception are formats/informats, in this case one file has to contain all definitions of formats/informats sharing the same name, e.g. numeric format `abc.`, character format `$abc.`, numeric informat `abc.`, and character informat `$abc.` all have to be kept in one file.
  - An object name is a file name, e.g. a definition of a macro named `%abc()` has to be contained in a file named `abc.sas`.
  - All files have to have `.sas` extension. Other files are ignored.
  - When the `function` type is used a definition of a function has to be enclosed in the following template of the `FCMP` procedure:

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

The `inlib=` and `outlib=` options are, literally, set to: "`work.&packageName.fcmp`" and "`work.&packageName.fcmp.package`" respectively. In case when functions from other packages are required to be used the `inlib=` option may be extended e.g. `inlib=(work.&packageName.fcmp work.OnePackagefcmp work.SecondPackagefcmp)`.

- When the `functions` type (mind the "s"!) is used a definition of a function *must not* to be enclosed in the `FCMP` procedure. In this case it has to be plain code of the function. All functions defined in the `..._functions` subfolder will be compiled with one `Proc FCMP` execution. An example of such "plain" subroutine code could be as follows:

```
subroutine increaseByOne(a[*]);
  outargs a;

  do i = 1 to dim(a);
    a[i] = a[i] + 1;
  end;
  return;
endsub;
```

- When type `format` is used then a definition of a format/informat has to be enclosed in the following template of the `FORMAT` procedure:

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

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

- When the `formats` type (mind the "s"!) is used a definition of a format/informat *must not* to be enclosed in the `FORMAT` procedure. In this case it has to be plain code of the format/informat. All format/informat defined in the `..._formats` subfolder will be compiled with one `Proc FORMAT` execution. An example of such "plain" format/informat code could be as follows:

```
value abc
  low -< 0  = "negative"
         0  = "zero"
  0 <- high = "positive"
      other = "missing"
;
```

- A definition of a `imlmodule` has to be plain code of the `IML` module. All modules defined in the `..._imlmodule` subfolder are compiled with one `Proc IML` execution and are stored in the `work.&packageName.iml` catalog.
- A definition of a `casludf` has to be plain code of the `CASL` user defined function. An example of such "plain" CASL UDF code could be as follows:

```
  function myFunction(x, y, z);
    result = x + y + z;
    return (result);
  end func;
```

All UDFs defined in all `..._casludf` subfolders are included with into `Proc CAS` by execution of single utility macro (see "User's utility macros for loading package content" subsection for details). UDFs definitions are stored inside the package and are loaded "on the fly" when the utility macro is called.

- A definition of a `proto` external C function has to be plain code of the `PROTO` C function. All functions defined in the `..._proto` subfolder are compiled with one `Proc Proto` execution and are stored in the `work.&packageName.proto` dataset. Function code has to contain the header, and the body of the function between `externc` and `externcend`, for example:

```
int doublePlusOne(int x);

externc doublePlusOne;
  int doublePlusOne(int x)
      return (2*x + 1);
externcend;
```

  If there are multiple directories with `PROTO` C functions then the content of each directory is marked by subsequent number in the `work.&packageName.proto` dataset with values: `package1`, `package2`, etc.
- `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 has to be inserted into a file inside one of the `exec` subfolders.
- `clean` folders are for cleaning after `exec`s, 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 has to be developed to remove that created object.

  Note: Each `exec` file should have `clean` file counterpart and vice versa. If the number of `exec` files and `clean` files differs but both are positive a warning is issued. But if `exec`s are positive and `clean`s are zero (or other way around) an error is issued!
- `kmfsnip` folders are dedicated for `kmf` (keybord-macro-file) files. Those files contain the code of KMF-abbreviation snippets which can be imported in SAS DMS or EG session. Structure of this type file is a bit special. It has the following form:

```
/*** HELP START ***//*
  <Part for help info notes.>
*//*** HELP END ***/

kmfCodeDesc: <The snippet single line description text.>

kmfCodeStart:
  <Code of the snippet,
   can be in multiple
   lines long.>
kmfCodeEnd:
```

  The `kmfCodeStart:` and `kmfCodeEnd:` special tags indicate (respectively) the beginning and the end of the snippet. Code between those two can be multiple lines long. The `kmfCodeDesc:` is a *single line* description text of the snippet. The name of the snippet is taken from the file name and is kept i lower case letters. During package loading the snippets are not load automatically (there is no such functionality). Instead the `packageName.kmf` file is generated in the `WORK` library location. Also an information note instructing how to import the snippets file is printed in the log. The code generating the `kmf` file was inspired by article [**8**].
- `ds2pck` folders are dedicated for the code of PROC DS2 packages. Structure of the file should be:

```
package PackageName / overwrite=yes;
  <... DS2 package code ...>
endpackage;
```

By default, if there exist a SAS data set (which is not a DS2 package file) a warning is issued and the package data set is not generated. To force overwrite, set the `DS2force=` parameter of the `%loadPackage()` macro to 1.

- `ds2thr` folders are dedicated for the code of PROC DS2 threads. Structure of the file should be:

```
thread ThreadName (<...parameters...>) / overwrite=yes;
  <... DS2 thread code ...>
endthread;
```

By default, if there exist a SAS data set (which is not a DS2 thread file) a warning is issued and the thread data set is not generated. To force overwrite, set the `DS2force=` parameter of the `%loadPackage()` macro to 1.

- `addcnt` folder (there can be only one such folder) contains all possible additional content which the Developer want to add to the package e.g., a `pdf` file with documentation, files with graphs, plots, and figures, an `html` page, a markdown `md` file, etc. Structure inside that folder can be arbitrary, it can contains files and subfolders, also nested. The content of the `addcnt` folder is transported into the package `zip` in binary form. Part of the code generating the additional content was inspired by article [**7**].
- 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 ***/`". The tags are not mandatory but if they are missing in a file a warning is printed into the log during package generation. If help tags are present in a file they have to be in proper order (start - end) and cannot be nested or overlap. If such situation takes place an error is printed into the log during package generation and the process is aborted. An example could be the following file containing a macro code, a help text, and other comment:

```
/*** HELP START ***/
/* >>> %ABC() macro: <<<
 *
 * Main macro which allows to do this and that...
**/
/*** HELP END ***/

/* macro definition */
/*** HELP START ***/
%MACRO ABC(
  param1 /* parameter 1 is used for ... */
 ,param2 /* parameter 2 is used for ... */
);
/*** HELP END ***/

    <... multi-line ...>
    <...    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 purpose:

```
/* >>> %ABC() macro: <<<
 *
 * Main macro which allows to do this and that...
 *
**/
%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 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 USECASE" section and "Appendix A" 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 `SPFinit.sas` file into this folder.

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

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

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

Developer's "under the hood". Before reading this subsection further we highly 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 %generatePackage() macro, a lot of things is happening behind the scenes. This section explains them in more details.
The first information the developer receives after the process ends is a summary report displaying basic information about the package content. In this summary the following elements are displayed: the package location (i.e. folder), developer's &sysuserid., creation timestamp, SAS version, the package encoding information (based on the description.sas file), and current SAS session encoding. From the description.sas file the package name, version, and license type are extracted and printed. Also the list of required elements is printed if any were provided. The last part of the summary is a table displaying a list of files used to build up the package.
But the summary is only the tip of an iceberg. The following steps are executed when the macro runs. At the beginning the description.sas file is tested for existence and when the result is positive the file is read otherwise process is stopped with error. The following macrovariables (descriptors): packageName, packageVersion, packageTitle, packageAuthor, packageMaintainer, packageEncoding, packageLicense, packageRequired (optional), packageReqPackages (optional) are created and obligatory ones (i.e. first seven) are tested for values. If at least one of the descriptors is missing the process is aborted with an error. If the package name is more than *24 characters* or contains illegal symbols (non alphanumeric or underscore) the process is aborted with an error too. Value of the package version should be in the major.minor.patch form or major.minor or major, where each component i.e., major, minor or patch is an integer, hence this is also tested.
If a zip file with package name exists inside the package folder the zip file is deleted and the new fileref is generated.
In the next step the package folder is scanned and structure of files and subfolders is extracted. Since files and subfolders with code have to be named only with lower case letters - it is tested, if the test fails the process is aborted with an error.
At this point the summary mentioned above is generated.
Further steps create so called "driving" files. The description.sas is copied into the zip file. The license.sas is either copied or MIT license is generated. The packagemetadata.sas file with descriptors macrovariables is created. The list of macrovariables generated by the code from the packagemetadata.sas contains the following variables: packageName, packageVersion, packageTitle, packageAuthor, packageMaintainer, packageEncoding, packageLicense, and packageGenerated. The firs seven contains values extracted from the description.sas file. The packageGenerated macrovariable contains timestamp when the package was generated in the ISO8601 form (YYYY-MM-DDThh:mm:ss).
At this point the iceloadpackage.sas file containing the %ICEloadPackage() "emergency" macro (described in the "User's "in case of emergency" section) is generated.
The next one is the load.sas file. If packageRequired or packageReqPackages macrovariables are present two parts of code for testing requirements are generated, respectively. Both codes are design to set up the packageRequiredErrors macrovariable to 1 (one) if requirements are not met. The first test compares the proc setinit output with provided list of required licensed SAS products. The second test compares the SYSloadedPackages macrovariable with provided list of required packages.

If a package from the list is not in the SYSloadedPackages then, to try to load required package, the %loadPackage() macro is called. If the packageRequiredErrors is positive the loading of the package is aborted with the following error message "ERROR: Loading package &packageName. will be aborted! Required SAS components are missing."

After requirements testing all "includes" are generated. In case of functions or formats/informats, if the first one is detected, a code to update cmplib or fmtsearch is generated. If the subfolders of the exec type are detected a snippet code to print their content is added. As the final part of the load.sas code for creating/updating the SYSloadedPackages global macrovariable is added.

After that a code snippet for loading "lazy datasets" is generated into a lazydata.sas code file.

The process continues with generation of the unload.sas code. As the first part the code to print and execute the clean type is assembled. As the second step the code for the macros and formats/informats deletion is generated. Deletion of functions follows after. And the last is the code for libraries unassignment and the SYSloadedPackages update.

In the third file, help.sas, following snippets are generated. The first to print out content of the description.sas file, namely the description part and list of required components if any are provided. The second to print out content of the license.sas file. The third snippet creates a data step used for content search and print out of the help text from the package files (macros, functions, etc.)

Content of the preview.sas file is similar to the content of the help.sas. The difference is that the code is printing out not the help portion of selected elements of the package but, to assure full and easy code transparency, prints out (into the log) all content of the file of a macro, a function, etc.

The final part of the %generatePackage macro is a series of data steps copying package code files into the zip file and testing existence, parity, and potential overlapping of help tags. If no tags are found a warning is printed in the log. If tags are mismatched or overlapping an error is printed.

Eventually within the package zip file we will find:

- Copies of all files from package subfolders but with modified names, what is needed to keep the ordering in place. Each code file name is extended with a prefix of a form: underscore, subfolder name, and dot. For example if a file name is abc.sas and a subfolder 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 the code executed by the %loadPackage() macro. The file content is built based on the subfolders and files structure provided by the developer. The file is a series of requirements tests, %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 the code executed by the %helpPackage() macro. The file contains 1) code which displays general package description, 2) code which searches for a content based on helpKeyword and prints out the information, and 3) code which, if helpKeyword is "License", prints out the license text.
- The preview.sas file containing the code executed by the %previewPackage() macro. The file contains 1) code which displays content of the description file, 2) code which searches for a content based on helpKeyword and prints out the the content of files found, and 3) code which, if helpKeyword is "License", prints out the license file.
- The unload.sas file containing the code executed by the %unloadPackage() macro. The file 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 at the beginning).

- The `addcnt.zip` file containing the *additional content* extracted by the `%loadPackageAddCnt()` macro. The file content is built based on the `addcnt` subfolder (if one was provided by the developer). This file is not used automatically during installation or loading a package. See the "Package additional content" to learn how to extract its content.

When the package `zip` file is ready, withe the help of the `HASHING_FILE()` function, a SHA256 hash digest of the package is generated. Two types digest are generated. One, with a prefix "`F*`", is generated based on the entire `zip` file using direct file location i.e.:

```
SHA256 = SHA256 = 'F*' !! HASHING_FILE("SHA256", pathname("&_PackageFileref_.",'F'), 0);
```

The other, with a prefix "`C*`", is generated based on the content of the `zip` file using file reference i.e.:

```
SHA256 = SHA256 = 'C*' !! HASHING_FILE("SHA256", "&_PackageFileref_.", 4);
```

In both snippets the `&_PackageFileref_.` macrovariable contains value of the fileref to the `zip`, but in the first one the value is resolved by the `pathname()` function.

As the "pre-final" part of the process the testing of the package is executed. Details about the testing process are provided in the next section.

The last code is executed optionally and runs when the `markdownDoc=` parameter is set to 1 (default is 0). The `markdownDoc` indicates if a markdown file named `packagename.md` with documentation be generated from help info blocks prepared by the developer. The process assumes that help info provided *is formatted according to the markdown rules* (see for example: `https://www.markdownguide.org/basic-syntax/` or `https://github.github.com/gfm/`). If for some reason developer want some of the files to be excluded when generating the documentation file, e.g., "internal functions" or "utility macros", it can be done by inserting an "exclusion comment" in the *first* line of the file. The "exclusion comment" can be one from the following list: `/*##DoNotUse4Documentation##*/`, `/*##ExcludeFromDocumentation##*/`, or `/*##ExcludeFromMarkdownDoc##*/`. They are case sensitive!
The last step of documentation generation was added for developers convenience. When the `packagename.zip` and the `packagename.md` are ready the `easyArch=` parameter set to 1 triggers creation a copy of the zip and markdown files with the version number. Basically the `packagename.zip` and the `packagename.md` are copied as the `packagename_XXX_.zip` and the `packagename_XXX_.md` where `XXX` is replaced by version number. This step is optional but was added so that developers could easy keep track or historical versions of the package.

```
..<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]
 |         +-xyz.sas [a file with a code creating macro XYZ]
 |
 +-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" sufix)]
 |            |
 |            +-efg.sas [a file with a code creating function EFG]
 |
 +-003_functions [mind the S at the end!, one file one function,
 |             |  only plain code of the function, without a "fcmp" header]
 |             |
 |             +-ijk.sas [a file with a code creating function IJK]
 |
 +-004_format [one file one format,
 |          |  option LIB= should be: work.&packageName.format
 |          |  (literally with macrovariable name and "format" sufix)]
 |          |
 |          +-efg.sas [a file with a code creating format EFG and informat EFG]
 |
 +-005_data [one file one dataset]
 |        |
 |        +-abc.efg.sas [a file with a code creating dataset EFG in library ABC]
 |
 +-006_exec [so called "free code", content of these files will be printed
 |        |  to the log before execution]
 |        |
 |        +-<if no file - folder may be skipped>
 |
 +-007_format [if your codes depend on each other 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]
 |
 +-008_lazydata [one file one dataset]
 |            |
 |            +-klm.sas [a file with a code creating dataset KLM in the WORK library
 |                       created on demand with the call: %loadPackage(packagename, lazyData=klm)]
 |
 +-010_imlmodule [one file one IML module, only plain code of the module]
 |             |
 |             +-abc.sas [a file with a code creating IML module ABC]
 |
 +-011_proto [one file one PROTO function, only plain code of the function]
 |         |
 |         +-pqr.sas [a file with a code creating externC function PQR]
 |
 +-012_casludf [one file one CASL user defined function, only plain code of the function]
 |           |
 |           +-fgh.sas [a file with a code creating CASL UDF function FGH]
 |
 +-013_ds2pck [one file one PROC DS2 package]
 |            |
 |            +-library.xyz.sas [a package LIBRARY.XYZ stored in LIBRARY.XYZ data set]
 |
 +-...<sequential number>_<type [in lower case]>
 |
 +-00n_clean [if you need to clean something up after an exec file execution,
 |         |  content of these files will be printed to the log before execution]
 |         |
 |         +-<if no file - folder may be skipped>
 |
 +-00n+1_test [code for testing the package, used only in the developer's session]
 ...
```

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

Developer's tests. When all elements of the package are ready and zipped into a `zip` file a good practice is to execute tests which will verify if and how the package works. This subsection describes what possibilities of testing are provided by the `%generatePackage()` macro and how to use them.
All tests are executed in clean and separate SAS sessions hence the `XCMD` option has to be turned on. A *separate session* means that a `systask` statement is used to call SAS binary and run new session in a batch mode. The `!SASROOT` folder is searched for SAS binary file as the default one. A *clean session* means that no configuration file (e.g. the one located in the `!SASROOT` folder) is used to run the testing session. The default behaviour can be altered with the following parameters of the `%generatePackage()` macro:

- the `sasexe=` points a *folder* where the SAS binary file is located.
- the `sascfgfile=` points either a *file* containing testing session configuration parameters, or if it is set to `DEF` value then the `!SASROOT/sasv9.cfg` is used.

By default the `%generatePackage()` macro executes test for loading, helping, and unloading a package. The `%loadPackage()`, `%helpPackage()`, `%previewPackage()`, and `%unloadPackage()` macros are called in such way that location of the package is the one provided by the `filesLocation` macroparameter. If there are any dependencies (i.e. additional packages are required), unless the `packages` fileref is assigned, the developer has to use the `packages=` macroparameter to point to the folder containing required packages and *also* a copy of the `SPFinit.sas` file. If the `XCMD` option is off then the `%generatePackage()` macro prints into the log code for testing loading.
Results of this test (and all other provided by the developer), i.e. log and listing, are stored in a subfolder of the maim session `WORK` library, the subfolder name has the following form: `test_YYYYMMDDtHHMMSS`. The default `WORK` folder can be altered with the `testResults=` parameter of the `%generatePackage()` macro which points developer's location where tests results should be stored.
When the test is done the log is scanned for any error or warning messages. As a summary the developer gets short output with a table presenting: exit status of the `systask` statement, value of the `sysrc` macrovariable after execution of the `systask`, the number of *error* messages from the log, and the number of *warnings* messages from the log.
If the developer wants to execute additional tests the following "files and folders" steps have to be performed:

- Create a subfolder of type `test` in the package folder, e.g `999_test`.
- Copy all files with the code for tests into the subfolder.

The test file does not require a code to load package, this code is provided in a separate `autoexec.sas` file so only the testing code should be in the file. There is no formal form of the test file but a good practice would be to call explicit error or warning if the test fails.
Since each file in the `test` subfolder is executed in a separate session the developer can chose to keep all tests in one file or splits tests into separate files.
Each test file invokes separate SAS session, to keep the `WORK` of that session the `delTestWork` parameter in the `generatePackage` macro should be set to 0.
To suppress any testing the developer has to set up the `testPackage=` macroparmeter to any value different than "Y", e.g. "N" is preferred.
To redirect location of testing results the developer has to set up the `testResults=` macroparmeter to a valid directory, e.g. `C:\testsresults` . Default value is null and means that the `WORK` is used.
The following list presents file references and libnames created during the testing process in the main developer's session: `sasroot`, `currdir`, `TEST`, `TESTWORK`.

Developer's laziness. A silent assumption in all previous sections was that the developer prepares a package content in separate files and organize those files in ordered directories. What if a developer has all the code prepared and tested but it is all in one file? Does the developer have to "cut & paste" everything by hand? Short answer is: No. But as it usually is, the devil is in details. A *utility macro* named

%splitCodeForPackage() can be helpful in situations when there is labor some process of creating separate files from one to do. The macro has two main parameters:

- the codeFile= points a *file* where "the-SAS-code-to-be-split" is located.
- the packagePath= points a *folder* where the package structure is generated.

The package structure is build based on a group of text *tags* added by the developer to the file with code. Those tags have very precise structure that is:

- "/*####$####-code-block-start-####$#### <tag spec> */" for the opening one and:
- "/*####$####-code-block-end-####$#### <tag spec> */" for the closing one

Tags are case insensitive. The "<tag spec>" inside tags can be a space-separated list of the form: "type1(object1) type2(object2) ... typeN(objectN)", but usually (99% cases) it will be just one, e.g., /*####$####-code-block-start-####$#### 001_macro(myMacroOne) */ and /*####$####-code-block-end-####$#### 001_macro(myMacroOne) */. The "<tag spec>" tells the macro into what files and folders given code snippet surrounded by a particular "start-end" pair of tags suppose to be copied.
For example when the %splitCodeForPackage() macro is executed on a file containing the following content:

```
/*##$##-code-block-start-##$## 01_macro(abc) */
%macro abc();
  % put I am "abc".;
%mend abc;
/*##$##-code-block-end-##$## 01_macro(abc) */

/*##$##-code-block-start-##$## 01_macro(efg) */
%macro efg();
  % put I am "efg".;
%mend efg;
/*##$##-code-block-end-##$## 01_macro(efg) */

proc FCMP outlib=work.f.p;
/*##$##-code-block-start-##$## 02_functions(xyz) */
function xyz(n);
  return(n**2 + n + 1)
endfunc;
/*##$##-code-block-end-##$## 02_functions(xyz) */
quit;
```

two directories (01_macro and 02_functions) are crated. The first one contains two files: abc.sas and efg.sas, and the second contains file xyz.sas.
Three special cases of "<tag spec>" are supported.

- When the "_all_(_all_)" tag is used a code snippet surrounded by such tag will be copied over to *all* files in *all* folders,
- When the "type(_all_)" tag is used a code snippet surrounded by such tag will be copied over to *all* files in a particular folder.
- When the "_all_(object)" tag is used a code snippet surrounded by such tag will be copied over to *all* files named "object.sas" in package folders.

When the process is executed dedicated summary tables are printed in the output window. If there are any problems during the execution, warnings or errors are printed in the log.
In the current version of the framework, if the process is executed multiple times files are *not* overwritten automatically. Each execution's content is "appended" to what already is in the packagePath= directory. If the developer want to start over, the packagePath= directory has to be manually purged from the previous execution's "leftovers".
Each file created with help of the %splitCodeForPackage() macro has the following line of comment: "/* File generated with help of SAS Packages Framework, version YYYYMMDD. */" added at the very first line of the file.

Note: Though the `%splitCodeForPackage()` macro is labeled as a utility one, consider it more like a "helping-hand" tool rather then "the-way-to-go" approach to packages programming.

GENERATING PACKAGE IN PRACTICE - a USECASE

The practical **example** will be build based on one of the author's favourite 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 "select" code within a data step, 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 name will be *SQLinDS* and it will be providing the `%SQL()` macro which will 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 built with 5 files: `description.sas`, two `macros`, one `function`, and one `library` (test files are not included). Let's assume we have created the following package folder `C:/SAS_PACKAGES/SQLinDS/` and we have copied the `SPFinit.sas` file into the `C:/SAS_PACKAGES/` directory. The structure of subfolders created for the package is presented in Figure 3.

```
..<C:/SAS_PACKAGES/SQLinDS/>
 |
 +-description.sas ❶
 |
 +-000_libname
 |           |
 |           +-dssql.sas ❷
 +-001_macro
 |         |
 |         +-dssql_inner.sas ❸
 |         +-sql.sas ❺
 |
 +-002_function
 |            |
 |            +-dssql.sas ❹
 +-999_test
 |        |
 |        +-test1.sas ❻
 |        +-test2.sas ❼
 |
 +-license.sas ❽
```

Figure 3: SQLinDS package - subfolders structure.

Details of the package files (including license) can be found in the "Appendix A" or on the web (see "TUTORIAL" section).

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(SPFinit.sas);
/*ods html;*/
%generatePackage(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 the approach presented above yourself and want to play a bit with the code and data you can download SAS programs 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`

Also the future versions of this article will be uploaded there.

TUTORIAL

If you are interested in learning how to work with SAS packages and the SPF visit the following tutorial:

`https://github.com/yabwon/HoW-SASPackages`

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] Frank Mittelbach, Michel Goossens, "The LaTeX Companion, Second Edition", Addison-Wesley 2004, ISBN 0-201-36299-6

[5] `https://realpython.com/python-modules-packages/` as of October 2019

[6] `https://www.internetblog.org.uk/post/1520/what-is-a-linux-package/` as of October 2019

[7] Kurt Bremser, "Talking to Your Host", WUSS 2022 Proceedings, `https://communities.sas.com/t5/SAS-User-Groups-Library/WUSS-Presentation-Talking-to-Your-Host/ta-p/838344`

[8] Tom Van Campen and Benny Haemhouts, "Dynamically generating macro invocations using SAS keyboard abbreviations", PhUSE 2012 Proceedings, `https://www.lexjansen.com/phuse/2012/cc/CC03.pdf`

ACKNOWLEDGMENTS

Author would like to acknowledge **Filip Kulon**, **Krzysztof Socki**, **Allan Bowe**, **Quentin McMullen**, **Piotr Wójcik**, **Michał Wojtasiewicz**, **Richard DeVenezia**, and **Christian Graffeuille** for their contribution!

CONTACT INFORMATION

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

`yabwon✉gmail.com`

or via the following LinkedIn profile:

`www.linkedin.com/in/yabwon`

Appendix A

The content of files composing the `SQLinDS` package.

❶

```
/* 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                                  :/*required, not null, constant value*/
Package: SQLinDS                               :/*required, not null*/
Title: SQL queries in Data Step                :/*required, not null*/
Version: 2.2                                   :/*required, not null*/
Author: Mike Rhoads (RhoadsM1@Westat.com)       :/*required, not null*/
Maintainer: Bartosz Jablonski (yabwon@gmail.com) :/*required, not null*/
License: MIT                                   :/*required, not null*/
Encoding: UTF8                                 :/*required, not null*/

Required: "Base SAS Software"                  :/*optional*/

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

# The SQLinDS package [ver. 2.2] ##############################

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

Copy of the article is available at:
https://support.sas.com/resources/papers/proceedings12/004-2012.pdf

Package provides ability to *execute* SQL queries inside a data step, e.g.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
  data class;
    set %SQL(select * from sashelp.class);
  run;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
See the help for the `%SQL()` macro to find more examples.

### Content ####################################################

SQLinDS package contains the following components:

1. `%SQL()` macro - the main package macro available for the User
2. `dsSQL()` function (internal)
3. `%dsSQL_inner()` macro (internal)
4. Library `DSSQL` (created as a subdirectory of the `WORK` library)

---
DESCRIPTION END:
```

❷

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

## >>> library `dsSQL`: <<< ########################

The `dsSQL` library stores temporary views
generated during the `%SQL()` macro execution.

If possible a subdirectory of the `WORK` location is created, like:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))/dsSQLtmp";
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if not possible, then redirects to the `WORK` location, like:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))";
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---

*//*** HELP END ***/

data _null_;
  length rc0 $ 32767 rc1 rc2 8;
  rc0 = DCREATE("dsSQLtmp", "%sysfunc(pathname(work))/"                );
  rc1 = LIBNAME("dsSQL",    "%sysfunc(pathname(work))/dsSQLtmp", "BASE");
  rc2 = LIBREF ("dsSQL"                                                );
  if rc2 NE 0 then
    rc1 = LIBNAME("dsSQL", "%sysfunc(pathname(work))", "BASE");
run;

/* list the details about the library in the log */
libname dsSQL LIST;
```

❸

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

## >>> `%dsSQL_Inner()` macro: <<< ##############

**Internal** macro called by `dsSQL()` function.
The macro generates a uniquely named SQL view on the fly
which is then stored in the `dsSQL` library.

Recommended for *SAS 9.3* and higher.

---

*//*** HELP END ***/

/* inner macro */
%MACRO dsSQL_Inner() / secure;
  %local query tempfile1 tempfile2 ps_tmp;
  %let query = %superq(query_arg);
  %let query = %sysfunc(dequote(&query));

  %let viewname = dsSQL.dsSQLtmpview&UNIQUE_INDEX_2.;

  %let tempfile1 = A%sysfunc(datetime(), hex7.);
  %let tempfile2 = B%sysfunc(datetime(), hex7.);

  filename &tempfile1. temp;
  filename &tempfile2. temp;

  %let ps_tmp = %sysfunc(getoption(ps));
  options ps = MAX;
  proc printto log = &tempfile1.;
  run;
  /* get the query shape i.e. the executed one */
  proc sql feedback noexec;
    &query
    ;
  quit;
  proc printto;
  run;
  options ps = &ps_tmp.;

  %put *** executed as ***;
  data _null_;
    infile &tempfile1. FIRSTOBS = 2; /* <- 2 to ignore header */
    file &tempfile2.;
    /* create the view name */
    if _N_ = 1 then
      put " create view &viewname. as ";
    input;
    put _infile_;
    putlog ">" _infile_;
  run;
  %put *****************;

  proc sql;
       %include &tempfile2.; /* the &query */
       ;
  quit;
  filename &tempfile1. clear;
  filename &tempfile2. clear;
%MEND dsSQL_Inner;
```

❹

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

## >>> ‘dsSQL()‘ function: <<< ####################

**Internal** function called by the ‘%SQL()‘ macro.
The function pass a query code from the ‘%SQL()‘
macro to the ‘%dsSQL_Inner()‘ internal macro.

Recommended for *SAS 9.3* and higher.

### SYNTAX: ##########################################################################
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
dsSQL(unique_index_2, query)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**Arguments description**:

1. ‘unique_index_2‘ - *Numeric*, internal variable, a unique index for views.

2. ‘query‘ -          *Character*, internal variable, contains query text.

---

*//*** 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 query 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;
```

❺
```
/*** HELP START ***//*

## >>> `%SQL()` macro: <<< ###########################

The **main** macro which allows to use
SQL queries in the data step.

Recommended for *SAS 9.3* and higher.

Based on the article *"Use the Full Power of SAS in Your Function-Style Macros"*
by *Mike Rhoads* (Westat, Rockville), available at:
https://support.sas.com/resources/papers/proceedings12/004-2012.pdf

### SYNTAX: ###################################################################
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
%sql(<nonempty sql querry code>)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The sql query code is limited to *32000* bytes.

### EXAMPLES: #################################################################

**EXAMPLE 1**: simple SQL query
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
data class_subset;
  set %SQL(select name, sex, height from sashelp.class where age > 12);
run;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**EXAMPLE 2**: query with dataset options
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
data renamed;
  set %SQL(select * from sashelp.class where sex = "F")(rename = (age=age2));
run;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**EXAMPLE 3**: dictionaries in the data step
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~sas
data dictionary;
  set %SQL(select * from dictionary.macros);
run;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---

*//*** HELP END ***/


/* Main User 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 query ***; /* print out the query 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;
```

❻

```
proc sort data=sashelp.class out=test1;
  by age name;
run;

data class;
  set %SQL(select * from sashelp.class order by age, name);
run;

proc compare base = test1 compare = class;
run;
```

❼

```
data class_work;
  set sashelp.class;
run;

data test_work;
  set %sql(select * from class_work);
run;

options dlcreatedir;
libname user "%sysfunc(pathname(work))/user";
%put *%sysfunc(pathname(user))*;

data cars_user cars_user2;
  set sashelp.cars;
run;

data test_user;
  set %sql(select * from cars_user);
run;

data test_user2;
  set %sql(select * from user.cars_user2);
run;

libname user clear;
%put *%sysfunc(pathname(user))*;

proc datasets lib = work;
run;
```

❽ 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.
```

Appendix B

Here reader can find headers (with short parameters description) of macros `%generatePackage()`, `%installPackage()`, `%loadPackage()`, `%unloadPackage()`, `%helpPackage()`, and others
When the framework macros are run *without* arguments or with `HELP` keyword, like: `%installPackage()`, `%loadPackage(HELP)`, `%helpPackage()`, `%unloadPackage(HELP)`, or `%generatePackage()` the help information (for all macros in the framework) is printed out into the SAS log. Since the `%listPackages()` has no arguments the help for the macro is printed when `%listPackages(HELP)` is run.

- ---

```
%macro generatePackage(
 filesLocation   /* location of package files
                    e.g. %sysfunc(pathname(work))/%lowcase(&packageName.) */
/* testing options */
,testPackage=Y   /* indicator if tests should be executed,
                    default value Y means "execute tests" */
,packages=       /* location of other packages if there are
                    dependencies in loading */
,testResults=    /* location where tests results should be stored,
                    if null (the default) the WORK is used */
,sasexe=         /* a DIRECTORY where the SAS binary is located,
                    if null (the default) then the !SASROOT is used */
,sascfgFile=     /* a FILE with testing session configuration parameters
                    if null (the default) then no config file is pointed
                    during the SAS invocation,
                    if set to DEF then the !SASROOT/sasv9.cfg is used */
,delTestWork=    /* indicates if ‘WORK‘ directories generated by user tests
                    should be deleted, i.e. the (NO)WORKTERM option is set,
                    default value 1 means "delete tests work" */
,markdownDoc=0   /* indicates if a markdown file with documentation
                    be generated from help info blocks */
,easyArch=0      /* when creating documentation file indicates if a copy of
                    the zip and markdown files with the version number in the
                    file name be created */
)/secure minoperator
```

- ---

```
%macro installPackage(
  packageName  /* space separated list of packages names,
                  without the zip extension */
, sourcePath = /* location of the package, e.g. "www.some.page/",
                  mind the "/" at the end */
, mirror = 0   /* indicates which location for package
                   source should be used */
, version =    /* indicates which version of a package to install */
, replace = 1  /* 1 = replace if the package already exist,
                  0 = otherwise */
, URLuser =    /* user name for the password protected URLs */
, URLpass =    /* password for the password protected URLs */
, URLoptions = /* options for the ‘sourcePath‘ URLs */
, loadAddCnt=0  /* should the additional content be loaded?
                   default is 0 - means No, 1 means Yes */
, SFRCVN =      /* name of a macro variable to store success-failure return code value */
)/secure
```

- ---

```
%macro verifyPackage(
  packageName                          /* name of a package,
                                          e.g. myPackage,
                                          required and not null  */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, hash =                              /* The SHA256 hash digest for
                                          the package generated by
                                          hashing_file() function, SAS 9.4M6 */
)/secure
```

```
%macro loadPackage(
  packageName                          /* name of a package,
                                          e.g. myPackage,
                                          required and not null  */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, options = %str(LOWCASE_MEMNAME)     /* possible options for ZIP filename */
, source2 = /*source2*/               /* option to print out details,
                                          null by default */
, requiredVersion = .                 /* option to test if loaded package
                                          is provided in required version */
, lazyData =                          /* a list of names of lazy datasets
                                          to be loaded, if not null then
                                          datasets from the list are loaded
                                          instead of a package, asterisk
                                          means "load all datasets" */
, zip = zip                           /* standard package is zip (lowcase),
                                          e.g. %loadPackage(PiPackage)
                                          if the zip is not available use a folder
                                          unpack data to "pipackage.disk" folder
                                          and use loadPackage in the form:
                                          %loadPackage(PiPackage, zip=disk, options=) */
, cherryPick=*                        /* space separated list of selected elements of the package
                                          to be loaded into the session, default value "*" means
                                          "load all elements of the package" */
, loadAddCnt=0                        /* should the additional content be loaded?
                                          default is 0 - means No, 1 means Yes */
, suppressExec=0                      /* indicates if loading of exec files
                                          should be suppressed, 1=suppress */
, DS2force=0                          /* indicates if PROC DS2 packages and threads
                                          should be loaded if a data set exists, 0=do not load
                                       */
)/secure
```

```
%macro loadPackageS(
  packagesNames /* A comma separated list of packages name,
                   e.g. myPackage, myPackage1, myPackage2, myPackage3
                   required and not null.
                   Package version, in brackets behind a package name,
                   can be provided, e.g.
                    %loadPackageS(myPackage1(1.7), myPackage2(4.2)) */
)/secure parmbuff
```

```
%macro unloadPackage(
  packageName                          /* name of a package,
                                          e.g. myPackage,
                                          required and not null  */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, options = %str(LOWCASE_MEMNAME)     /* possible options for ZIP filename */
, source2 = /*source2*/               /* option to print out details,
                                          null by default */
, zip = zip                           /* standard package is zip (lowcase),
                                          e.g. %unloadPackage(PiPackage)
                                          if the zip is not available use a folder
                                          unpack data to "pipackage.disk" folder
                                          and use unloadPackage in the form:
                                          %unloadPackage(PiPackage, zip=disk, options=)
                                       */
)/secure
```

```
%macro helpPackage(
  packageName                          /* name of a package,
                                          e.g. myPackageFile.zip,
                                          required and not null  */
, helpKeyword                          /* phrase to search in help,
                                          when empty prints description
                                          "*" means print all help
                                          "license" prints license */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, options = %str(LOWCASE_MEMNAME)      /* possible options for ZIP filename */
, source2 = /*source2*/                /* option to print out details,
                                          null by default */
, zip = zip                            /* standard package is zip (lowcase),
                                          e.g. %helpPackage(PiPackage,*)
                                          if the zip is not available use a folder
                                          unpack data to "pipackage.disk" folder
                                          and use helpPackage in the form:
                                          %helpPackage(PiPackage, *, zip=disk, options=) */
, packageContentDS = 0                  /* indicates if a data set with package
                                           content should be generated in WORK,
                                           if set to 1 then WORK.packageName_content
                                           dataset is created
                                         */
)/secure
```

```
%macro listPackages()/parmbuff
```

```
%macro previewPackage(
  packageName                          /* name of a package,
                                          e.g. myPackageFile.zip,
                                          required and not null  */
, helpKeyword                          /* phrase to search for preview,
                                          when empty prints description
                                          "*" means prints all */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, options = %str(LOWCASE_MEMNAME)      /* possible options for ZIP filename */
, source2 = /*source2*/                /* option to print out details,
                                          null by default */
, zip = zip                            /* standard package is zip (lowcase),
                                          e.g. %previewPackage(PiPackage,*)
                                          if the zip is not available use a folder
                                          unpack data to "pipackage.disk" folder
                                          and use previewPackage in the form:
                                          %previewPackage(PiPackage, *, zip=disk, options=)
                                         */
)/secure
```

```
%macro extendPackagesFileref(
 packages /* A valid fileref name,
             when empty the "packages" value is used. */
)/secure
```

```
%macro loadPackageAddCnt(
  packageName                          /* name of a package,
                                          e.g. myPackage,
                                          required and not null  */
, path = %sysfunc(pathname(packages)) /* location of a package,
                                          by default it looks for
                                          location of "packages" fileref */
, target = %sysfunc(pathname(WORK))   /* a path in which the directory with
                                          additional content will be generated,
                                          name of directory created is set to
                                          `&packageName._AdditionalContent`
                                          default location is SAS work */
, source2 = /*source2*/               /* option to print out details,
                                          null by default */
, requiredVersion = .                 /* option to test if loaded package
                                          is provided in required version */

)/secure
```

```
%macro splitCodeForPackage(
 codeFile          /* a code file to split */
,packagePath=      /* location for results */
,debug=0           /* technical parameter  */
,nobs=0            /* technical parameter  */
)
```