SESUG 2025 - Paper HOT-012

SAS Packages - a Modern Code-Sharing Medium for SAS®

Bartosz Jabłoński - yabwon / Warsaw University of Technology

ABSTRACT

When working with SAS code, especially with complex code, there is a moment when you decide to break it into small pieces. You create separate files for macros, formats/informats, for functions or data, too. Eventually the code is ready and tested, and sooner or later you will want to share it with another SAS programmer. You have developed a program using local PC SAS, but the deployment is on a Viya server running on a Linux OS. Your code is complex (with dependencies such as multiple macros, formats, datasets, etc.) and is difficult to share. Often when you try to share code, the receiver will quickly encounter an error because of a missing macro, missing format, or whatever...small challenge, isn't it? In the article I discuss a solution to the problem - the idea of SAS packages, an answer to a reflection: *Modern data focused languages, like R or Python, have vast ecosystems for building packages. Those environments allow their users to share their ideas, inventions, and code in an easy, almost seamless way. Why did't SAS, with its profound and historically well-established impact on data analysis, embrace such a marvelous idea?* The article covers the following topics: what are SAS packages, how to use and develop them, and how to make code-sharing a piece of cake, and of course, what opportunities, possibilities, and benefits SAS packages bring to the community of SAS programmers?

Table of contents

COMPETITIVE ADVANTAGE LOST

SAS as a data processing and analytic tool and language has a profound and well-established history. It is an undeniable fact. When one looks at its "market share" in terms of *my tool of choice* among data analysts, statisticians, data scientists, etc. in recent years, we see that languages like R or Python took over a significant portion of that area. Why is that? The first, and most popular, argument cried out in all those "SAS vs. R" or "SAS vs. Python" battles happening on the Internet is: "*For SAS you have to pay and the <(insert a name)> is for free*". I dare to say, despite used the most often, this is not the point. I am not saying it is not valid argument; I am just saying it is not *the* main reason. When you think about any mature analytic system of enterprise level it *always* comes with a price. You will always have to pay. The location of the cost is just different. With SAS, you pay for the tool and the people who can help to support you come with it. With

Insert "R" or "Python".

<(insert a name)>, you get the tool free of charge but you will have to pay for those people who can help to support you...

My observation is the following: it is a community of people sharing (in nearly "real time") their work, discoveries, and solutions through packages is *the* competitive advantage that <(insert a name)> has over SAS. Whenever the core part of <(insert a name)> is short with some functionality, there is almost always someone who provides a package covering that gap, sometimes even multiple packages. To name a few examples: Python does not support rectangular data structures (I mean data frames) out of the box, no worries the `pandas` package is here to help. For many years base R did not support process piping, no worries the `magrittr` package provided `%>%` operator. Data frames in R could be handled "better" (e.g., faster, more comfortable, memory efficient, etc.) no worries we can try the `data.table` or the `tibble` package. One needs to draw a plot for an analysis in Python, no worries, the `matplotlib` or the `seaborn` are there for us, etc.

You may argue those examples are invalid because handling data frames (data sets for SAS) is what SAS does out of the box. You can say the DATA step itself is a form of a pipe, and furthermore that SAS has the SGPLOT procedure (among other graphing tools) in it. Yes that is true; there is a group of smart people developing and adding new functionality to SAS, but that group is finite; it has its own time limitations and work-power constrains and as a result there is functionality that SAS could have but is missing... Some of those things are, for example, the latest statistical methods, not implemented only because they were published just after the latest SAS release. Some of those things are not "missing" per se; they are doable in SAS, but the implementation requires very advanced programming skills and complex program structure that both could be easily enveloped through SAS package interface.

To give you more insight on how deeply the idea of sharing through packages is "soaked" into the <(insert a name)> community, let me tell you an anecdote. It is the story of how the GIGS package was created for SAS. The other day Allan Bowe pointed me to a post at `communities.sas.com` titled: "Looking for experienced SAS package developer" published February 2nd, 2024[1]. The post's author, Bancy from *The London School of Hygiene & Tropical Medicine*, was asking: "Our team is looking for a SAS programmer to work on a package for assessment of fetal, neonatal, and child growth." The team was a group of statisticians rooted in R programming style, and by that, their first question was not: "*is it possible to build a package in SAS?*", their "natural" expectation was that it *is* doable, and they were just looking for someone who knew how to do that...

I am convinced that introduction of a code-sharing culture based on packages can elevate the SAS community to the next level. In this article I will present the idea of SAS packages; I explain a tool that facilitates the idea, i.e., the SAS Packages Framework; and I will advocate for using and developing SAS packages in the SAS programming community.

BUT WHY?

You may ask: Why introduce a new tool since there are SASAUTOS or compiled catalogs? Well, there are several advantages; let's count some of them:

- Both SASAUTOS and compiled catalogs are mainly dedicated to macros (the later also for formats); a package allows sharing macros, user-defined functions, formats, informats, IML modules, even data, or DS2 and CAS-L code, and many more!
- In contrary to compiled catalogs, packages can be shared among different operating systems, and they keep the source code at their fingertip, so it can be (easily) previewed on demand and cannot be lost.
- In contrary to SASAUTOS, a package combines all code needed for proper execution into one file, and that code is compiled and executed in proper designed-by-developer order.

[1] See: `https://communities.sas.com/t5/Developers/Looking-for-experienced-SAS-package-developer/m-p/914197##M6467`

- In contrary to plain files in SASAUTOS, a packages binds all its code under version number; it has a generation timestamp embedded in it; and provides a unique SHA256 digest that facilitates easy verification whether the package was modified by some "man in the middle" or not.
- Packages provide easily accessible documentation that can be printed in the SAS session LOG at a user's request. Everything is at your hand, with no need for Internet access to work.
- SAS packages, when designed properly, can be deployed and used on every SAS environment you desire: local Base SAS, remote SAS9 servers with Enterprise Guide, SAS Viya with SAS Studio, and even the latest SAS Workbench with Visual Studio Code.

That is just the beginning. A much longer list of reasons promoting the use of packages as a modern code-sharing medium can be found in [**Jablonski 2024**]. Now, let's not waste our time on advertising why "*the honey is better than something else...*" and let's have some fun with technicalities. Just to clarify, this article is introductory and aims to covers the "most basic cases". I will not dive into all details; I will focus on presenting that "most popular use cases" (the 80/20 rule). There are other resources (mentioned through this article) where details are discussed and decomposed to the bare metal, without mercy, presenting the reader all the gory bits.

In the next section I will look at the tool that allows to use and develop SAS packages, i.e. the SAS Packages Framework. And after that, I will preview some of available features for playing with packages.

SAS PACKAGES FRAMEWORK

The SAS Packages Framework (SPF for short), first introduced in [**Jablonski 2020**], is a collection of SAS macros (currently a dozen of them) that allows to build and use SAS packages. The framework facilitates all tools required for installation, loading, getting help, validating, and of course generating SAS packages. The SPF is open source, MIT licensed, 100 percent written in SAS, with no need for any third-party tools, and is hosted at the following GitHub repository:

https://github.com/yabwon/SAS_PACKAGES

There are several ways to install it, but I will cover here only the most popular ones. Very detailed tutorial materials, explaining all the nuts and bolts, can be found in [**Jablonski 2023**].

INSTALL SPF

Regardless you will be installing SPF on your local laptop or on a company server (to the delight of the SAS folks) the step "zero" is to create a directory where the SPF and packages (in the future) will be stored. To use packages in a SAS session, there is, in fact, only one requirement: The packages directory *has to be referred* as `packages`, and that reference cannot be deleted during the session.

For example, if you decided the packages directory is `/sas/packages/`, in the SAS session run:

code: assign fileref

```
filename packages "/sas/packages/";
```

When: 1) the directory exists, 2) the `packages` fileref is assigned, and 3) the SAS session has the Internet access (to the GitHub repository of the framework in particular), the following code snippet will install the framework on the computer:

code: installation code

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

%installPackage(SPFinit);

filename SPF clear;
```

Successful execution of the installation should have a note similar to the following in the LOG:

the log - successful installation

```
INFO: Source path is https://github.com/SASPAC/

INFO: Calling: SPFinit

### SPFinit() ###
*** spfinit start ****************************************

[...]

 Installing the SPFinit package
 in the /sas/packages directory.
 Done with return code rc=0 (zero = success)
*** spfinit end ******************************************

INFO: Package SPFinit installed.
```

You need to execute this process only for the very first time. If the SPF file is on the computer and you, for example, want to install the latest version of the framework - do it exactly as you would install any other package (see next section).

If access to the Internet from the SAS session is not satisfied, it is possible to execute the installation manually. All you need to do is: visit the `https://github.com/yabwon/SAS_PACKAGES` repository; navigate to subdirectory named `SPF`; and download file `SPFinit.sas` to the `/sas/packages/` directory. That is all; the installation is done.

ENABLE SPF

Similarly to how the libraries are assigned at the beginning of our SAS session, the SAS Packages Framework, too, must be enabled. This can be done by running the following snippet:

code: enable the framework

```
filename packages "/sas/packages/";
%include packages (SPFinit.sas);
```

After running those two lines the SPF's macros are compiled and the framework is ready for work. That code snippet can be executed at the start of a session, but personally, I prefer to save it in my `autoexec.sas` file, so I do not have to run it every time myself.

As it was mentioned earlier, the SPF is composed of twelve macros (for now). Ten of them are dedicated for users:

- `%listPackages()`,
- `%installPackage()`,
- `%verifyPackage()`,
- `%helpPackage()`,
- `%previewPackage()`,
- `%loadPackage()`,
- `%loadPackageS()`,
- `%loadPackageAddcnt()`,
- `%unloadPackage()`,
- `%extendPackagesFileref()`

and two additional are dedicated to developers:

- `%generatePackage()`,
- `%splitCodeForPackage()`.

Their purpose and functions will be discussed in the subsequent sections. If you have SPF enabled in the SAS session, to get help notes about those macros all you need to do is to run them with `HELP` keyword as an argument, e.g., `%listPackages(HELP)`. This will give a brief preview of their parameters printed in the LOG.

SAS PACKAGES

Packages are code bundles created for users by users (called *developers* in this case). A semi-formal definition that is usually provided follows:

Definition: SAS PACKAGE

A **SAS package** is an automatically generated, single, stand alone `zip` file containing organized 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 allows: 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 the deployment (installation) process.

The essence of a package is to combine all code the developer wants to share, contained and organized (in the order the developer wants) into a single file, with additional driving code to facilitate all those features that a tool like a package would be expected to have or provide (e.g., loading, getting help, etc.)

SEARCHING FOR PACKAGES

The first question to be asked, after finding out what packages are, is: "Where can we find packages?" The answer is layered, but the following paragraphs summarize it.

Currently, there are two officially supported public SAS packages repositories. They are SAS Packages Archive (SASPAC for short) and PharmaForest. The first one, SASPAC, is the framework's default target location for searching packages. SASPAC is set up under the following link:

`https://github.com/SASPAC`

SASPAC ia a dedicated place for all sorts of publicly available packages (as long as they satisfy certain quality standards) created for solving various tasks and needs. We can say it is an "industry-agnostic" repository, dedicated to all SAS packages users.

The second one, PharmaForest, organized by the PHUSE Japan Open Source Technology working group, is located under:

`https://github.com/PharmaForest`

It is focused on pharmaceutical industry SAS programmers.

The other locations depend heavily on your (yes, your!) activity. If you decide to build a package:

- You can share it yourself among other users on your own. After all, all you need to do to be able to use a package in your SAS session is to save the `packagename.zip` file in the packages directory.
- You can publish it on GitHub and use the SPF for installation. In this case, the assumption is that you create a repository named the same as the package name, i.e., `packagename`, and that the branch on which the `packagename.zip` file is located in called `main` (see next section).
- You can contact SASPAC or PharmaForest teams and ask for setting a repository for your package there.

I WANT A PACKAGE

The installation process for a package, in a nut shell, is all abut downloading packages `zip` files into the packages directory on your computer (or the computer you want to use SAS packages on). The most popular ways to do it include:

- For packages located in SASPAC just run:

code: install from SASPAC

```
%installPackage(packageName)
```

The macro will search through SASPAC as the default location, and if it finds the package there, it will download it to the packages directory. The SASPAC located packages allow the `%installPackage()` macro to install historical versions too.

- For packages in PharmaForest, use the `mirror=` parameter:

code: install from PharmaForest

```
%installPackage(packageName, mirror=PharmaForest)
```

- If you decide to publish a package on GitHub, the framework supports the following syntax:

code: install from GitHub

```
%installPackage(packageName, GitHub=GitHubUserName)
```

It will work as long as your setup satisfies assumptions mentioned in the previous section.

- For other network locations, you can do:

code: install from arbitrary network location

```
%installPackage(packageName, sourcePath=https://webpage.com/hosting/place/)
```

- Eventually, if you cannot do it coding, as already mentioned, a package can be installed just by manually copying the `packagename.zip` into the packages directory.

SHOW ME THAT PACKAGE

The basic instinct, after package installation, is to load and use it, but... Patience! When you gain more experience, loading a package will be the first thing you do (probably even by "autoexecing" the process). But for the first time, let's have some protection. For the beginning let's take a look at the package, see what it can offer, and determine if it is "safe and valid". The SAS Packages Framework provides three macros that deliver the "taste it before take it" functionality.

The first macro is about getting help information. Content offered by a package (i.e., macros, functions, formats, data, etc.) can be displayed with the `%helpPackage()` macro. When you run it:

code: basic help info

```
%helpPackage(packageName)
```

basic help information provided by the developer is displayed in the LOG. Additionally, names and types of all elements of the package are printed. Furthermore, if any required SAS components or some other package dependencies exist, they are displayed. Finally, a note about SPF's version (that was used to generate that package) is shown.

If you are interested in displaying help information for any particular component, you simply run the `%helpPackage()` macro with the component name as the second argument:

code: help info about particular element

```
%helpPackage(packageName, componentOfInterest)
```

If the developer provides help notes in form of a markdown text, then also a `packagename.md` file can be created automatically when the package is generated. The markdown-formatted help file (if it exists) can be downloaded during package installation. Markdowns are pretty convenient and aesthetic when used with the SAS Workbench VSCode environment.

The second macro in this little threesome is the %verifyPackage() macro. When a package is generated by the developer, the zip file is created and the SHA256 checksum of that zip is calculated. The checksum guarantees verification. It ensures that no one altered, modified, or broke the package file on the road between the developer and the user. You run the %verifyPackage() macro with the package name as the first argument and and the checksum as the second argument (hash= parameter). Information in the LOG displays an error when the verification fails.

The third macro in the game is %previewPackage(). It gives a convenient way to preview the source code of a package. It works using the same principles as the help macro. The source code is displayed in the LOG.

TAKE ADVANTAGE OF A PACKAGE

When you know that the installed package is valid and what it has to offer, you can finally play with it. To start using what the SAS package provides, first you have to load it. The duo of macros dedicated to that purpose is %loadPackage() and %loadPackageS() (and the second one is a convenience wrapper for the first one).

The %loadPackage() macro takes a package name as the first parameter and in 99 percent of cases, it is run as simply as the following snippet:

code: load package to SAS session

```
1 %loadPackage(packageName)
```

The process under the hood extracts package content one after another, compiles macros, functions, formats, generates data (if they are present), DS2 packages[2], IML modules, CAS-L functions, etc. All of those steps are clearly logged to keep the process transparent and traceable. Created content is stored mainly in the WORK library. When SAS creates formats or functions and they already exist, SAS by default informs the user in the LOG. For macros it is the framework that makes such checks. The framework also automatically updates search paths for functions (CMPLIB) and formats (FMTSEARCH). Furthermore, if there are any package dependencies the %loadPackage() macro checks if those packages are already loaded, and for those that are not, it triggers loading.

There are several optional parameters available to alter %loadPackage() macro behavior, but let's mention just two of them - the two most popular ones. The requiredVersion= parameter allows a user to request package loading only if a particular (e.g., high enough) version is accessible. If you are picky about components of a package, you can use the cherryPick= parameter to select only those components you want or need. This feature needs a comment! If you decide to cherry-pick a package, not all loading steps are executed so the result of the process is by design not equivalent to full, regular loading.

The final step of loading a package is the creation (or value refreshment) of a special technical macro variable &SYSLOADEDPACKAGES. that contains the list of already loaded packages with their versions.

At this point, you have the SAS session "enriched" with the package content, and you are ready to use whatever the package offers. Enjoy!

As mentioned earlier, the second macro, %loadPackageS(), is a syntactic-sugar wrapper over the %loadPackage() macro. The %loadPackage() macro allows a user to load one package at a time, and with the %loadPackageS() macro (the "s" stands for several) we can list multiple packages for loading at once, but at a cost of little less flexible parametrization.

[2]The idea presented in this article should not be confused with other occurrences of "package" concept which can be found in various places of the SAS ecosystem e.g., Proc DS2 packages, SAS/IML packages, SAS ODS packages, SAS Integration Technologies Publishing Framework packages, or event an Enterprise Guide *.egp file.

The `%listPackages()` is another handy macro. When you are not sure (or you simply do not remember) what packages you have, just run:

code: list available packages

```
%listPackage()
```

and all packages stored at your disposal in locations affiliated to the `packages` fileref will be displayed in the LOG.

CLEAN UP

When everything you wanted to do is done and you are satisfied with the package work, you can clean up the SAS session by remove all of the package's components generated on loading from the session. Unloading is as simple as running the following snippet:

code: clean after a package

```
%unloadPackage(packageName)
```

The framework takes care of everything and reverse the SAS session to the status from before loading. A note here! For the unloading process to be 100 percent successful the developer must pay attention when creating content of the `exec` type files. The code in the `exec` type case requires some additional care (see articles in the REFERENCES for details).

LET'S MAKE A LITTLE PACKAGE

When you want to build a package, the hardest part is to write its content (i.e., all that programs to be shared with other users). But that inevitable part, the development, is also the most fun part! Taking your programs and adjusting them (so that an "average John Smith" can use it) by adding help notes, parameter verification, and writing test cases, etc. It can be a very educational and satisfying process. When the code is ready, the remaining part (i.e., generating a package), in its principles, is almost an analogy of: $1)$ putting labeled documents into folders, $2)$ ordering those folders in a drawer, and $3)$ adding a *description* note on the drawer front. The leading actor of a package-generation process is the `%generatePackage()` macro. As "eerie" as the process may sound when seen for the first time, there is high (say, nine cases out of ten) probability that you will run the macro the following way:

code: generate package

```
%generatePackage(filesLocation = /path/to/package/content
                 ,markdownDoc = 1)
```

After the execution, remember to examine the LOG and the summary printed in the output window to see how the generation went!

As mentioned earlier, the aim of this article is to introduce the SAS packages. If you are eager to build your own package, resources describing the process with tons of examples are available in article [**Jablonski 2021**] and in tutorial materials separately available here [**Jablonski 2023**]. Both resources are designed to describe package-creation process in a step-by-step fashion and I encourage you to read them.

BE A STARGAZER

This article is a manifesto describing the idea of SAS packages as a code-sharing medium. Every new idea, including SAS packages, needs endorsement, support, and promotion. Invaluable to the endeavor are the so-called "early adopters" - those are, the people who decide to spare some of their time to play with the idea, to experiment, and to enrich the "infancy" niche with their feedback. That is why I encourage you to engage! For proactive, if you have more resources (aka free time and skills) at your disposal, try to learn how to build packages and share your work with the community. If you have fewer resources, try to experiment with existing packages. Remember to share your feedback with developers.

Regardless your available resources, visit the https://github.com/yabwon/SAS_PACKAGES repository and give it a star (⭐) to increase repository visibility at GitHub so more people can learn about SAS packages. If you are LinkedIn user, you can join *SAS Packages Users & Developers* group (SPUDs). Eventually, if you think the idea of SAS packages is beneficial, please visit communities.sas.com and up-vote this New Ballot Idea post at SAS Product Suggestions channel:

https://communities.sas.com/t5/SAS-Product-Suggestions/Add-SAS-Packages-Framework-to-the-SAS-Base-Viya/idi-p/815508

requesting to add SAS Packages Framework to the SAS9 and Viya.

CONCLUSION

In the article, I have discussed the idea of sharing SAS programs and solutions that are created by users for users through SAS packages. The SAS Package Framework is a tool designed and developed to use and generate SAS packages. I have shared references for further reading to expand your subject-matter knowledge. You have also learned fundamentals of using SAS packages in the SAS session. Now, let's make a punch line for this article! It has been based on a well known and popular "Expanding Brain" meme[3] (aka "Galaxy Brain"):


Write ad'hoc code whenever there is a task to do.


Write reusable and parameterized code for that task!


Wrap that code in a macro in sasautos or in catalog!!


Build a SAS package!!!

The End

REFERENCES

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

[Jablonski 2021] Bartosz Jabłoński, "My First SAS Package - a How To",
SAS Global Forum 2021 Proceedings, 1079-2021
https://communities.sas.com/kntur85557/attachments/kntur85557/proceedings-2021/59/1/Paper_1079-2021.pdf
also available at: https://github.com/yabwon/SAS_PACKAGES/tree/main/SPF/Documentation/Paper_1079-2021

[Jablonski 2023] Bartosz Jabłoński, "Share your code with SAS Packages a Hands-on-Workshop",
WUSS 2023 Proceedings, 208-2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-208.pdf

[Jablonski 2024] Bartosz Jabłoński, "Use of SAS Packages in the Pharma Industry – Opportunities, Possibilities, and Benefits",
PHUSE 2024 Proceedings, SM01-2024, https://www.lexjansen.com/phuse/2024/sm/PAP_SM01.pdf

ACKNOWLEDGMENTS

The author would like to acknowledge the group of enthusiastics who continuously support the idea of sharing SAS code through SAS packages, the members of the PHUSE Japan Open Source Technology working group: Hiroki Yamanobe, Yutaka Morioka, and Ryo Nakaya!
The author would like to acknowledge Troy Martin Hughes whose linguistic contribution made this paper look and feel as it should!

[3] See https://knowyourmeme.com/memes/galaxy-brain

CONTACT INFORMATION

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

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

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

Appendix A - code coloring guide

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

- The code snippets use the following coloring convention:

code: is surrounded by a black frame

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

- The LOG uses the following coloring convention:

the log - is surrounded by a blueish frame

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

INDEX