PharmaSUG 2026 - Paper AP-124

SAS® Packages - an Ask-About-Anything Game

Bartosz Jabłoński - yabwon / Warsaw University of Technology

ABSTRACT

Modern data-focused languages, like R or Python, have vast ecosystems for building packages (i.e., reusable, structured, and efficient code containers). Those environments allow their users to share their ideas, inventions, and code in an easy, almost seamless way. Why haven't SAS, with its profound and historically well-established impact on data analysis, embrace such a marvelous idea? If this or similar questions about SAS packages bothers your sleep, fear no more! This article's purpose is to answer them and give you a solid foundation for future work with SAS Packages.

Table of contents

WHAT IS A SAS PACKAGE?

Packages are code containers created for users by users (called *developers* in this case) to share SAS code in a convenient and practical way. A semi-formal definition that I usually provide goes as 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 to provide (e.g., loading, getting help, etc.). All this is done according to the "K.I.S.S." principle[1] to minimize possible confusion.

WHERE ARE PACKAGES LOCATED?

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 default target location for searching packages. SASPAC is set up under the following link:

`https://github.com/SASPAC`

SASPAC is 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. You can say it is an "industry-agnostic" repository, dedicated to all SAS packages users.

The second one, PharmaForest, organized by the *PHUSE Data Visualization & Open Source Technology* working group (started in PHUSE Japan OST), 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 a package on GitHub and use the SAS Packages Framework for installation (see the packages installation question).
- You can contact SASPAC or PharmaForest teams and request a repository for your package.

DO I HAVE TO MAKE IT PUBLIC?

The idea behind SAS packages is to become a code sharing standard for SAS, but it is not said you have to share created packages "publicly." If your organization is interested in creating SAS packages but want to keep them internal, there is nothing wrong in that and there is no problem with that since installation of

[1] K.I.S.S. - Keep It Simple Stupid

packages located in private GitHub repositories is fully supported. I personally took part in creating packages that were internally distributed inside a company but never made public.

Whichever approach - public or private - you decide to use, the important thing is that with SAS packages you share your code in a standardized format that makes the process much more convenient.

HOW ARE SAS PACKAGES BETTER THAN SASAUTOS OR COMPILED CATALOGS?

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 latter also for formats); a package allows sharing macros, user-defined functions, formats, informats, IML modules, even data, or DS2 and CAS-L code, and much 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.
- In contrary to plain files in `SASAUTOS`, a package 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 during some man-in-the-middle attack 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 in 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 cover 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.

ARE THERE ANY TOOLS NEEDED TO USE AND BUILD SAS PACKAGES?

Yes, the tool that allows you to create and use SAS packages is the *SAS Packages Framework*.

WHAT IS THE SAS PACKAGES FRAMEWORK?

The SAS Packages Framework (SPF for short), first introduced in [**Jablonski 2020**], is a collection of SAS macros (currently more than a dozen of them) that allow a programmer 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 to run.

The SAS Packages Framework is composed (for now) of a group of macros classified in the following way: nine of them are dedicated for using packages:

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

two are dedicated for developing packages:

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

and six are utility macros:

- `%extendPackagesFileref()`,
- `%relocatePackage()`,
- `%isPackagesFilerefOK()`,
- `%bundlePackages()`,
- `%unbundlePackages()`,
- `%SasPackagesFrameworkNotes()`.

Of course the "classification" is not strict; it is just a convention. The purpose and functions of those macros 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., `%SasPackagesFrameworkNotes(HELP)`. This will give a brief preview of their parameters printed in the LOG.

WHERE IS THE SPF LOCATED?

The SAS Packages Framework file, i.e. `SPFinit.sas`, with all the documentation and examples, is hosted at the following GitHub repository:

`https://github.com/yabwon/SAS_PACKAGES`

IS IT SAFE TO USE SPF?

Yes!

The SAS Packages Framework is open source and 100 percent written in SAS. The framework is provided in the form of a plain `.sas` file. Every aspect of the tool can be investigated and the source code can be previewed with any text editor.

The SPF itself needs no third-party tools to run. All you need is a SAS session, does not matter which: Base, Viya, or Workbench.

There are several ways to install the SPF, but in the essence it all boils down to downloading the `SPFinit.sas` file on your computer. Very detailed tutorial materials, explaining all the nuts and bolts, can be found in [**Jablonski 2023**].

WHAT SETUPS ARE SUPPORTED BY SAS PACKAGES FRAMEWORK?

The SAS Packages Framework works under Windows, Linux, and UNIX operating systems. Supported SAS releases are Base SAS (9.4M3 and higher), Viya, and Workbench. User interfaces where it can run are: DMS, Enterprise Guide, SAS Studio, and Visual Studio Code. Basically any available mix of the above operating systems, SAS releases, and interfaces is supported.

IS IT HARD TO INSTALL SAS PACKAGES FRAMEWORK?

Regardless of whether you will be installing SPF on your local computer 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 *must be referred to* 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

```
1 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 produce 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 upcoming questions).

If access to the Internet from the SAS session is not guaranteed, 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.

HOW TO ENABLE THE SPF?

Similarly to how the libraries are assigned at the beginning of your 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.

ARE THERE ANY PLACES I CAN LEARN ABOUT IT?

Articles discussing SAS packages are for example: [**Jablonski 2020**], [**Jablonski 2021**], or [**Jablonski 2024**].

There are a few YouTube recordings presenting packages with varying level of details, starting a general overview and ending with a "let's build a package" tutorials. Visit the "Recordings and Presentations" section of the SAS Packages Framework's `README.md` file located in the SPF's GitHub repository.

Extensive training materials are there too! To learn more visit:

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

DO I NEED A GITHUB ACCOUNT TO WORK WITH PACKAGES?

No, you do not need Github account! The SPF, the tutorial, and packages in SASPAC and PharmaForest are public repositories, so you can access them even without a GitHub account.

Though, if you consider support (see upcoming questions) having GitHub account will make it easier and more effective.

IS IT HARD TO BUILD A PACKAGE?

The answer is both "Yes" and "No." Why is that, you may ask? The process of creating a package has two natural stages, one is harder the other is easy.

The first step is to create the package's content (i.e., all the code you want to share with others). This part may be a laborious task, especially when you want to do it right. Creating a package with solid input checks, a high-level of generality, proper tests, and outstanding documentation may require dedication and hard work. That is why my answer was "Yes, it's hard." But...

When your code is polished and ready to be packaged, the second step, package generation, with the SPF's help, is almost trivial. An analogy that fairly well represents this step is the following: generating a package from an earlier prepared code is like: 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 the package-generation process is the `%generatePackage()` macro. The process may look "eerie" when seen for the first time, but there is a high probability (say, nine in ten cases) that you will run the macro the following way:

code: generate package

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

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

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

WHERE TO LOOK FOR SUPPORT?

I have already mentioned the training materials, but if you need more help do not hesitate to contact me directly. Also, if you are a LinkedIn user, you can join the *SAS Packages Users & Developers* group (SPUDs).

HOW CAN I SUPPORT THE PROJECT?

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. The most basic

engagement you can do is to make "noise" about the idea of SAS packages and the SPF. Talk, talk, talk about them! Show them to your team-mates; convince your manager to build internal company packages. If you are a proactive SAS community member, and you have more resources (aka free time and skills) at your disposal, try to learn how to build packages and share your work with other members of the community. If you have fewer resources, try to experiment with existing packages, play, test, and remember to share your feedback with developers!

Regardless of 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. Eventually, if you find the idea of SAS packages 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.

WHAT IS THE GOOD SHAPE OR SIZE OF A PACKAGE?

There is no such thing as a good shape or size. There exist packages containing only one or two components (e.g., macros or functions), and at the same time you can find packages with hundreds of elements. It all depend on the package purpose. Tailor your package to suit the need the best.

WHAT THOSE MACROS IN SPF DO?

Every package, except the code provided by the developer, contains so-called driving files. They are additional files added during the generation process to the package to allow the framework to do the job of loading, getting help notes, listing, or cleaning up after a package. The macros provided by the framework pull those driving files and execute the code they contain.

WHAT IS THAT PACKAGE INSTALLATION ALL ABOUT?

The installation process for a package, in a nut shell, is all about 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)
```

In this case mirror=PharmaForest and mirror=3 can be used interchangeably.

- 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 some additional assumptions. In this case, by default, the framework assumes that you created a repository named the same as the package name, i.e., packagename, that the package file is located in top level, and that the branch on which the packagename.zip file is located in is called main. Also, as long as you configure a fine-grained personal access GitHub token, you can use private repositories.

- 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 by coding, a package can be installed just by manually copying the `packagename.zip` into the packages directory.

ARE THERE ANY POST-INSTALLATION STEPS?

When you gain more experience, loading a package will be the first thing you do (probably even by "autoexecing" the process) after installation. But for the first time, it is a good practice to 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 functionality.

First, getting help information. Content provided by a package (i.e., macros, functions, formats, data, etc.) can be displayed with the `%helpPackage()` macro. When you run:

code: basic help info

```
%helpPackage(packageName)
```

basic help information written 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 a particular component (e.g., a function), 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 shares help notes formated as a markdown text, then also a `packagename.md` file can be created automatically when a package is generated. The markdown-formatted help file (if it exists) can be downloaded during package installation. Markdowns are pretty convenient and aesthetic, for example, when used with the SAS Workbench VS Code environment.

The second macro 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 the checksum as the second argument (`hash=` parameter), for example:

code: package verification

```
%verifyPackage(packageName, hash=F*ABCDEF1234567890ABCDEF1234567890ABCDEF123456)
```

Information in the LOG displays an error when the verification fails, e.g., because someone was messing up with the package code or content. When the SHA256 check sum is not provided, the macro calculates it on its own (for information purpose), and additionally prints the SHA1 and MD5 checksums in the LOG.

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

WHAT HAPPENS WHEN A PACKAGE IS LOADED?

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. There are two macros dedicated to that purpose: the `%loadPackage()` and `%loadPackageS()` macro (and the second one is a convenient 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 reaches the `load.sas` driving file under the hood and extracts package content one after another; it compiles macros, functions, and formats; it 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 about it 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.

WHAT PACKAGES DO I HAVE ACCESS TO?

When you are not sure (or you simply do not remember) what packages you have installed on your machine the `%listPackages()` is a handy macro, just run:

code: list available packages

```
1 %listPackages()
```

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

[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 even an Enterprise Guide `*.egp` file.

WHAT SHOULD I DO WHEN I HAVE FINISHED USING A PACKAGE?

When everything you wanted to do is done and you are satisfied with the package work, you can clean up the SAS session by removing 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

```
1 %unloadPackage(packageName)
```

The framework takes care of everything and revert 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).

WHAT ARE BUNDLES?

A **SAS packages bundle** is a `zip` file containing one or more SAS packages and the bundle's metadata files inside it. Such a SAS bundle can be used as a container for a "snapshot" of the current SAS packages state in the SAS environment.

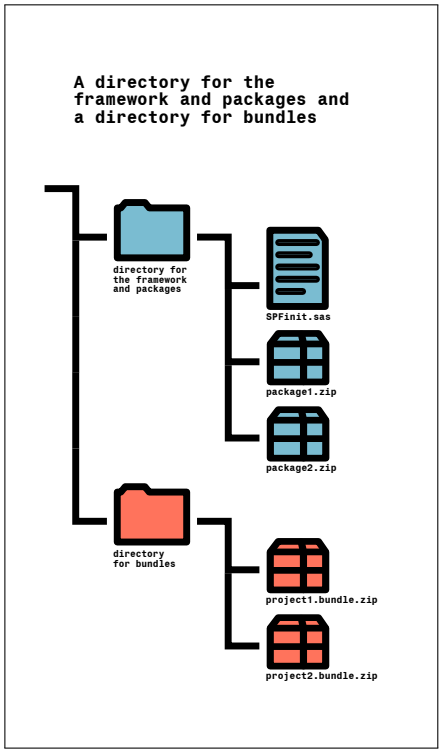


Figure 1. Bundles can store snapshots of your packages environment.

In a single user environment, like local PC SAS or environments where users have their own separate space (e.g., SAS Workbench), keeping the packages directories tidy and well organized may not necessarily be a "highly required" demand, but in a setup where multiple users or project teams share a centralized location for packages, the environment (e.g., packages versions) cannot be readily changed. Keeping the environment tidy can save programmers from a so-called "packages inferno"in which no one knows which version is where, which one is used, and "why my project is no longer working after a packages update?"

The `%bundlePackages()` and `%unbundlePackages()` macros (both build atop the `%relocatePackage()` macro) serves that purpose. The first macro allow to create a bundle of packages; the second extracts packages from a bundle.

When your project needs a fixed version of packages for execution you can, for example, create a packages environment "snapshot" by saving those packages inside a bundle at the end of the project and later, if need be, unbundle them (to a project-specific packages directory) so that you have the versions that are required.

CAN I LOAD ONLY PART OF A PACKAGE?

Yes, you can! The cherry-picking (`cherryPick=` option in the `%loadPackage()`) allows that. If a package has 42 components and you need only two elements, say `factorX` and `compoundV`, you can run:

code: cherry-picking

```
1 %loadPackage(packageName, cherryPick=compoundV factorX)
```

and only those two will be loaded. The list should be a space-separated list of components' names. One note though. An old saying goes: "There is no such thing as a free lunch;" in case of cherry-picking, it also

applies. When a package is cherry-picked, not all standard steps of the loading process are triggered. Since the cherry-picking selects only a part of the package the `&SYSLOADEDPACKAGES.` macro variable is not updated. Packages from the `ReqPackages` list (if present) are not loaded automatically (you have to load them yourself). The `%unloadPackage()` macro executed on a partially loaded package may sometimes issue (irrelevant) warnings, e.g., "Subroutine XYZ cannot be found for deletion."

WHAT IS LAZYDATA AND WHEN SHOULD I USE IT?

When a package provides data sets (i.e., code generating them), by design, data sets are loaded automatically when the `%loadPackage()` macro runs. The `lazydata` type provides an alternative to that behavior. The `lazydata` type also carries code for data sets, but those are not automatically loaded during the loading process. User has to explicitly "ask" for them. One may ask why to have lazy data sets at all? One example will be the following: lazydata can be very handy for demonstration of packages features, but user may not want to have them loaded to the SAS session every time. By providing data as the `lazydata` type the data sets are generated only when user directly requests for loading them.

If a package contains lazydata you can load them manually by calling the `%loadPackage()` macro with the `lazyData=` parameter, just like that:

code: loading lazy data

```
%loadPackage(packageName, lazyData=lazyDataSetName)
```

When you want to load multiple data sets use a space-separated list of names, and to load *all* lazy data sets at once use asterisk (`*`).

HOW DO I INSTALL A SPECIFIC VERSION OF A PACKAGE?

The SAS Packages Archive stores packages with their historical versions. When you want to install a particular historical version of a package, e.g., the latest version is `2.3.4` and you want version `1.2.3`, by default the `%installPackage()` macro will give you the latest version... But, by calling the macro like this:

code: install historical version

```
%installPackage(packageName(1.2.3))
```

you can ask for installation of the `1.2.3` version. The requested version should be provided in parentheses just after package name, and no spaces should be added.

In the same way, you can install any historical version of the SAS Packages Framework. Just remember that the installation of historical versions can reduce capabilities of the tool (earlier versions may not have all the features the latest one provides). In general, I always recommend using the latest version of the SPF. It not only gives you the latest bells and whistles but also has the most recent error fixes applied.

WHAT IS THE SYSLOADEDPACKAGES MACRO VARIABLE AND HOW CAN I USE IT?

When a package is loaded to a SAS session the last step of the process is the creation or update of the `&SYSLOADEDPACKAGES.` macro variable. The `&SYSLOADEDPACKAGES.` macro variable is a technical macro variable containing a list of all packages loaded to the current SAS session. You can use it to see if a given package (and with which version) was loaded. It can also be a handy tool for session auditing, e.g., when you need to keep track what packages were used during SAS session processing. Simply run

code: print loaded packages

```
%put &=SYSloadedPackages.;
```

CAN I USE PACKAGES WITHOUT INTERNET ACCESS?

Yes, absolutely! As long as you are able to copy a package `zip` to your packages directory you are good to go. Everything you need (including help notes) for a package to comfortably work is contained inside that `zip`! Internet access is only needed when you want to install packages from online repositories, like SASPAC or PharmaForest. After you are done with the installation you need no Internet access whatsoever.

CAN I USE PACKAGES WITHOUT SPF ON MY MACHINE?

As you already know, getting the SPF to your computer is a trivial thing. But even if you do not have the SPF on your machine, packages created with the SAS Packages Framework are still accessible. They can be "self loaded." This is called an "In Case of Emergency" loading (ICE-loading) or a cold loading. All you need to do to have a package working for you is to run code similar to the following snippet:

code: In Case of Emergency loading

```
filename ice ZIP "<directory/containing/packages>/somepackage.zip";
%include ice(iceloadpackage.sas);
filename packages "<directory/containing/packages>/";
%ICEloadPackage(packageName1)
%ICEloadPackage(packageName2)
...
```

This type of loading only loads package content. This means that, since you do not have SPF's macros, you will not be able to read help notes, verify the package, or unload it, etc.

CAN I USE THE AUTOEXEC.SAS TO HAVE PACKAGES ALWAYS AVAILABLE?

Of course you can! In fact, this is the way I load packages to my SAS session.

In my autoexec.sas I have added the following snippet:

code: Bart's autoexec.sas

```
[...]
filename packages "/home/bart/packages";
%include packages(SPFinit.sas);

%loadPackageS(SQLinDS, DFA, macroArray, BasePlus, GSM, bpUTiL, evExpress)
```

This way, whenever I start a brand new SAS session, I have my favorite packages available.

In SAS Studio you can add those lines in the window that pops up after selecting `Edit Autoexec File`. In the Enterprise Guide you can navigate to `Tools` ↦ `Options` ↦ `SAS Programs` and mark and edit the "Submit SAS code when server is connected." If you do it smartly (e.g., by dynamic packages directory selection), you can have packages loaded automatically on whatever session you are connecting to.

Eventually, if your SAS admin team is cool, they can set and add a "global/organization level" SAS packages directory and enable the framework automatically. Users would only need to load packages they want.

CAN I HAVE PACKAGES STORED IN MULTIPLE LOCATIONS?

Since the SAS packages fileref is assigned through the `FILENAME` statement it can point to multiple directories on your computer. The SAS Packages Framework supports multiple scenarios for packages location. Both a single and multi-directory setup is possible (see the Figure 2).

The simplest setup is when you have both the SAS Packages Framework file (i.e., the `SPFinit.sas`) and all your packages in a single directory. This is the setup I recommend in 99 percent of cases, but let's for example assume your organization has an official location for SAS packages, say `/sas/packages`. You are an active SAS packages developer and you have generated a package for your team. You can store it

someplace, say `/home/myteam/packages`, and use it together with the "official" location by running the following:

code: multiple locations

```
filename packages ("/sas/packages" "/home/myteam/packages");
```

This way, in the current SAS session, your directory will be appended at the end of the `packages` fileref.

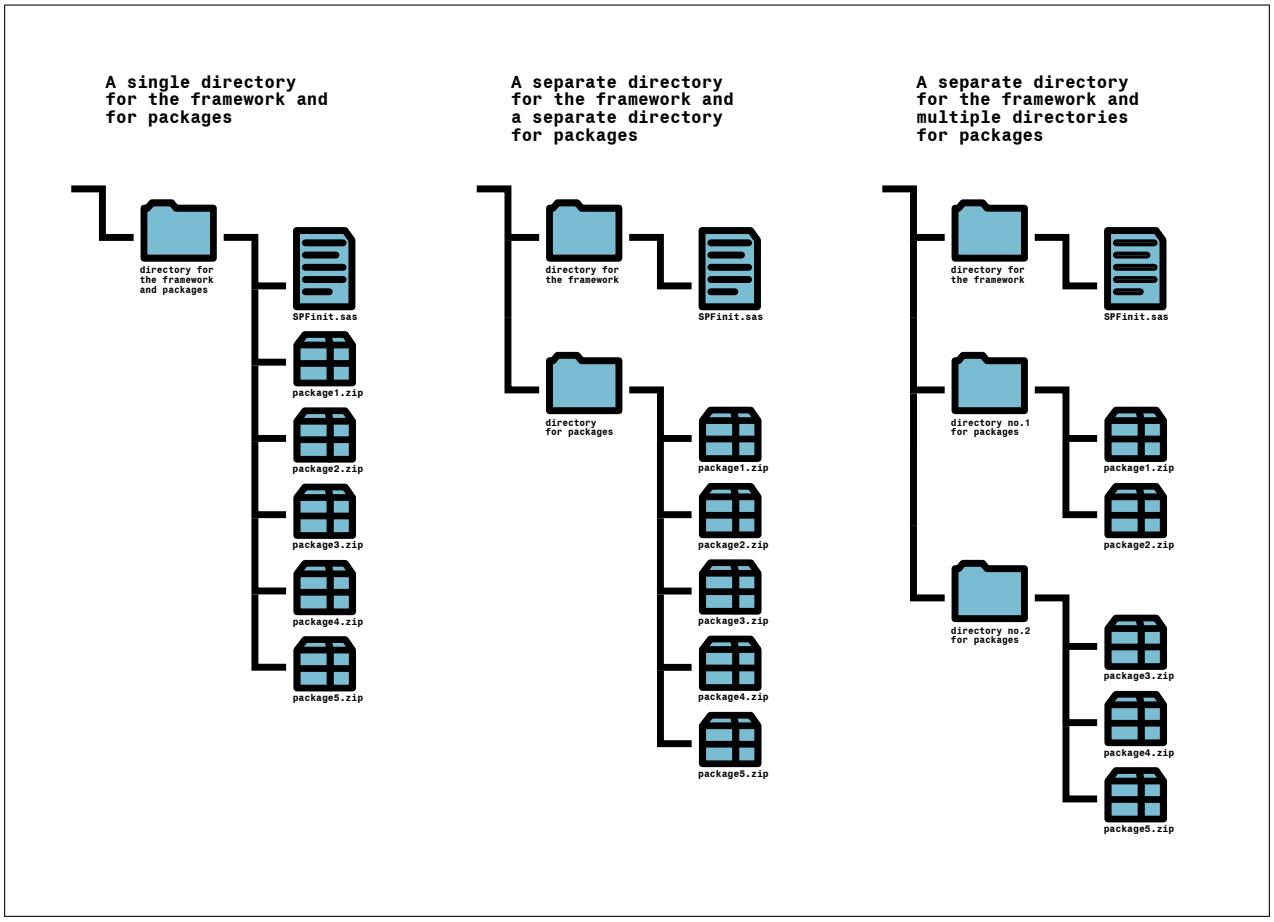


Figure 2. Examples of supported setups for packages location.

As mentioned, a setup with multiple directories is a good idea if you are developing packages or, for example, working on multiple projects and use different packages for them. But what do you do when your organization has a central location for packages and the framework, and your SAS admins team (as I mentioned earlier) starts the framework and assigns `packages` fileref "autoexecally", and you want to avoid hardcoding the "official" path next to your private location for packages? In such a case, even when the `packages` fileref is predefined, you can use the `%extendPackagesFileref()` macro to easily alter the `packages` fileref for your session by running the following code:

code: altering packages fileref

```
filename packages ("/your/location" %extendPackagesFileref());
```

WHAT HAPPENS IF TWO PACKAGES IN MY PACKAGES FILEREF HAVE THE SAME NAME?

When there are multiple packages in multiple directories, the rule that the SAS Packages Framework follows is similar to the `LIBNAME` statement behavior. When you are loading a package, the SAS Packages Framework traverses the `packages` fileref paths, searches from left to right, and selects the first occurrence of the package. When you are installing a package, the framework uses the first path on the `packages` fileref list

for the installation location. If you decide to load (install) a package from (to) a different directory, all you need to do is to add that directory as the first directory in the `packages` fileref list.

In other words, when you expand the `packages` fileref using the `%extendPackagesFileref()` macro (like in the example from the previous section), during your current SAS session, the `"/your/location"` directory will be used for packages installation. If there exists a package in the `"/your/location"` directory, named the same as an existing package in one of the paths of the original `packages` fileref, your package will be used first for loading. Of course, for multi-directory setup, you have to be "more SAS session aware!"

WHAT TYPES OF CODE CAN A PACKAGE CONTAIN?

Currently, the SPF supports 12 (or 14, counting "s"-es) standard code types and two "special" types. Those standard types are the following:

- `libname` for code generating libraries assignments,
- `macro` for macros,
- `function` and `functions` (mind the "s" at the end) for `PROC FCMP` functions,
- `format` and `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 code generating data sets,
- `lazydata` for code generating data sets that will be loaded on demand (lazy loading data sets),
- `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` thread.

If you want to share your code you will have to split it into snippets that fit into one of those categories. Also since these are "standard" ones, the SPF knows how to clean them from the SAS session during package unloading.

In case the code you want to share does not fit into any of the preceding categories[3] you can use special types:

- `exec` for so-called "free code",
- `clean` for the code cleaning the session after execs.

The `exec` is designed to be a "temporary" container for code that will not fit in the listed categories or for a code that will be executed when the package is loaded (see the DFA package at SASPAC to see example of such case). Since it is non-standard code and the framework does not know how it should be unloaded from the session, the `clean` type is how the developer can explicitly instruct SAS how to clean up after `exec`.

If, for example, your package provides a macro and a data set, then you have the `macro` type and the `data` type code. Next, you create the `macro` directory and the `data` directory in the source code directory, and you put the file containing macro code inside the `macro` directory, and you put the file containing data set code inside the `data` directory. It is as simple as that.

To be clear, the list above is not exhaustive. The framework is developing and expanding, so some new types can be added in the future if need be.

[3] Theoretically, everything in SAS can be embedded into a macro, i.e., `%macro test(); <..anything..> %mend;`, but this is *not* the assumption we want to operate and build our programs on.

HOW DO I ADD DOCUMENTATION TO MY PACKAGE?

If you are eager to read the answer to this question, it means you are a high-quality developer, after all: "if it's not documented, it does not exist..."

The first place where you can add help notes about your package is the `description.sas` file (see Appendix C - example of the `description.sas` file). All the text provided in the `description.sas` file that is enclosed between the `DESCRIPTION START:` and the `DESCRIPTION END:` tags is used as a part of the general help note.

The next place where you can add help notes is directly in code files. When you are developing a package component (e.g., a function, or a macro) the text snippets (comments) or code parts you want to display when `%helpPackage()` macro executes should be surrounded between special comments-tags that the SPF recognizes. Those tags are `/*** HELP START ***/` and `/*** HELP END ***/`. Here is an example of a small macro program with added help notes tags:

code: setting tags for help notes

```
/*** HELP START ***//*
This is the '%myLittleMacro()' macro.
It does some cool things with data.

Syntax:
~~~~~~~~~~~~~sas
%myLittleMacro(
  <dataset>
)
~~~~~~~~~~~~

Parameters:
  1) dataset - required, an input data set.

Macro declaration is:
~~~~~~~~~~~~~sas
*//*** HELP END ***/

/*** HELP START ***/
%macro myLittleMacro(dataset) / secure;
/*** HELP END ***/

 <some cool code goes here...>

 <even cooler code goes here...>

%mend myLittleMacro;

/*** HELP START ***//*
~~~~~~~~~~~~
Example 1. Basic use case:
~~~~~~~~~~~~~sas
  %myLittleMacro(sashelp.class)
~~~~~~~~~~~~

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

With help notes tags set this way the `%helpPackage()` macro would produce the following text in the SAS session LOG:

the log - help notes printed

```
This is the '%myLittleMacro()' macro.
It does some cool things with data.

Syntax:
~~~~~~~~~~~~sas
%myLittleMacro(
  dataset
)
~~~~~~~~~~~~

Parameters:
  1) dataset - required, an input data set.

Macro declaration is:
~~~~~~~~~~~~sas
%macro myLittleMacro(dataset) / secure;
~~~~~~~~~~~~

Example 1. Basic use case:
~~~~~~~~~~~~sas
  %myLittleMacro(sashelp.class)
~~~~~~~~~~~~
```

As you probably noticed, setting the comment opening (`/*`) directly after the help-start tag and the comment closing marker (`*/`) directly before help-end tag, like in this example:

code: setting tags for help notes

```
/*** HELP START ***//*
comment text
*//*** HELP END ***/
```

improves the printout aesthetics. Basically, because comment markers are in the same line as help tags, they are not printed and the only text displayed in the LOG is nice-looking "`comment text`". This type of formatting is not a formal requirement; the SPF prints out everything between help tags lines, but it makes help notes look elegant and gives an interesting opportunity described in the next section.

CAN I GENERATE A MARKDOWN DOCUMENTATION FILE FROM MY PACKAGE?

As you may notice, the help notes in the previous question's example looks like plain text with some fancy formatting. That is true; this formatting is a markdown document style formatting[4]. If you pay attention and take care to the way your help notes are formatted, it will give you two following features: 1) they will still look nice when displayed as a plain text; 2) they can be used to generate a nice-looking markdown documentation file that can be distributed alongside your package file.

All you need to do to generate such a markdown document containing help notes and documentation is to run the `%generatePackage()` macro with the `markdownDoc=` parameter. If you do so, the SPF will: crawl over all package files and extract help notes; combine them in a single file; add a header, table of contents, and even the license note at the bottom. The result will be a professional-looking documentation file that can be shared with package users. Of course it will look good, as long as you decide to use proper formatting aligned to the markdown rules. The framework is "naive" and when the `markdownDoc=` parameter is used the SPF does not verify if the formatting is correct; it basically assumes you did your job right, and runs. If you were sloppy, the created result will reflect that.

[4] See the following link to learn about markdown formatting more: `https://www.markdownguide.org/cheat-sheet/`

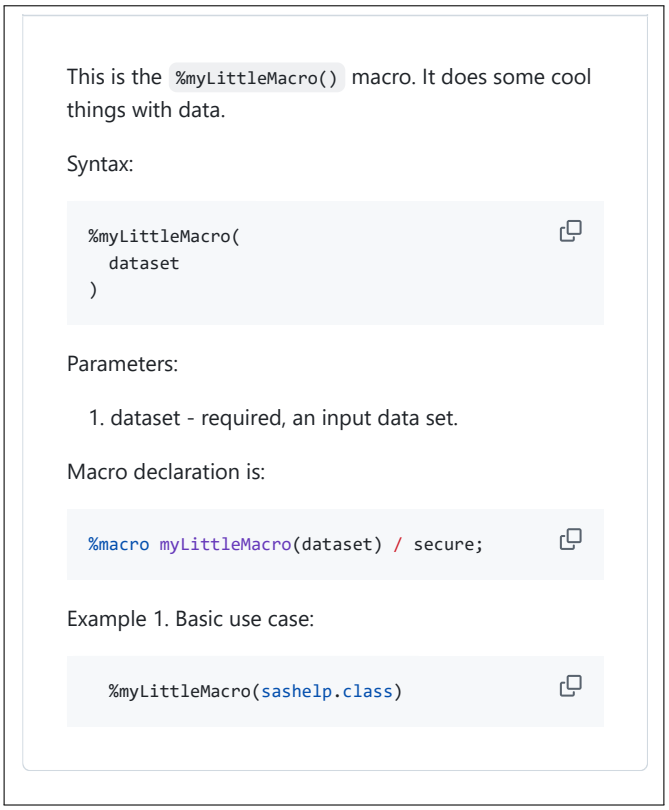
In case you would like to exclude some files' help notes from the markdown documentation file, e.g., files of the package internal functions with which the user never interacts, you can do it by adding one of the following markers:

code: exclusion markers

```
/*##DoNotUse4Documentation##*/
/*##ExcludeFromDocumentation##*/
/*##ExcludeFromMarkdownDoc##*/
```

in the *very first line* of the file you want to exclude.

Here is an example of how a markdown file rendered from the help note presented in the previous section may look like:



Definitely looks appealing!

HOW DO I TEST MY PACKAGE BEFORE RELEASING IT?

To run package tests you can ask the SPF to run them for you. The SAS Packages Framework testing mechanism is based on external SAS sessions running code. In other words, for every code file containing a test, the framework takes your test code file and runs it in a brand new SAS session. Therefore, it requires the `XCMD` option to be enabled.

For SPF to run your tests, you need to do the following: 1) inside your package source code directory create a subdirectory named `test`; 2) put every test in a separate sas file, e.g. `test1.sas`, `test2.sas`, etc.; 3) run the `%generatePackage()` macro. The macro automatically checks if the `XCMD` is on, traverses through `test` directory, picks tests one by one and runs them. The framework takes care of including the package loading before your test runs, so you do not have to add loading code to the test file. The SPF also triggers one automatic test that checks the package loading process. The test is executed as the first one and verifies if the package loads in the first place. Finally, when all tests are executed the SAS Packages Framework scans every test's log and prepares a summary of the execution.

The location where tests results (logs, listings, etc.) should be stored can be provided, so all testing artifacts can be saved for later audit.

HOW DO I CONTROL THE LOADING ORDER OF THE CODE?

When you asked "is it hard to build a package?" I have said that package organization is similar to putting documents into folders and those folders into a labeled drawer. But how should those folders be ordered?

Let's say you have a package that contains an FCMP function, a format, and a macro. And let's say the macro uses the format and the format uses the function. So to work properly they will have to be loaded in proper order, function first, format next, and macro last[5].

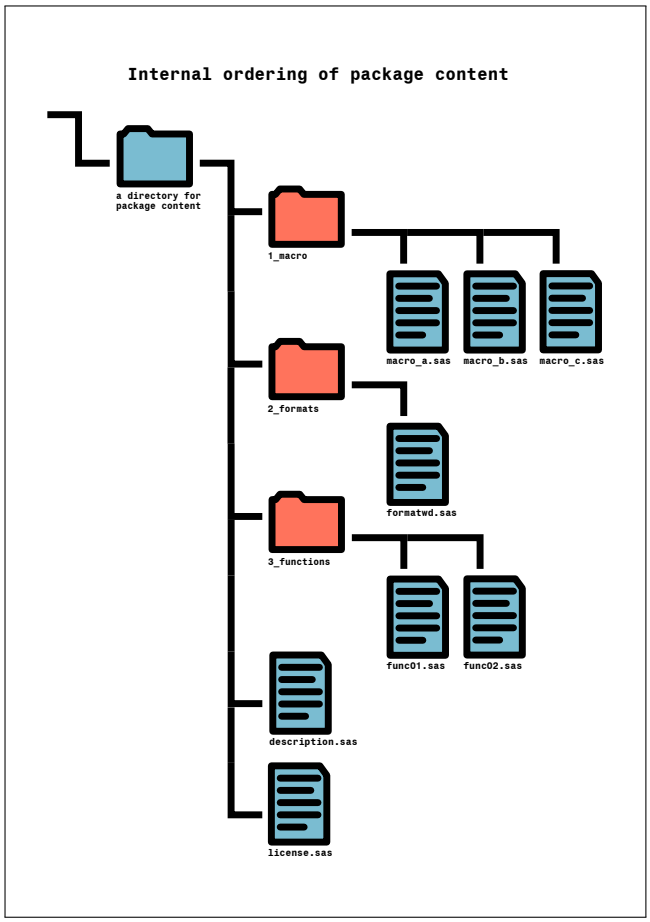


Figure 3. Ordering package content.

You already know that different code types should be located inside dedicated type directories. All you need to do to order them the way you want is to add numerical prefixes, for example: `001_functions`, `002_formats`, and `003_macro`.

If you have multiple programs of the same type but you want to load them in particular order you can use prefixes for that too, for example with: `01_macro`. `02_macro`, and `03_macro`, first all macros from the `01` will be loaded, next all from the `02`, etc.

Figure 3 shows an example of how ordering of the package's content directory can be organized. In this case the developer forces loading order so that macros are loaded first, the format is next, and functions are loaded last.

WHAT IS THE `%SPLITCODEFORPACKAGE()` MACRO?

When you start with a brand new package, from scratch, you can easily organize the code. Every macro, function, IML module, etc. goes to a separate file, and every file goes to a directory of dedicated type. Then you run the `%generatePackage()` macro and you are good to go, but... Sometimes life is tricky and it may happen that you will not start from scratch but from a file (or a few) that has all the code in it. Since the code is one undivided "blob" you have to manually do splitting between files and directories... Lucky for you, the SAS Packages Framework offers the `%splitCodeForPackage()` macro that helps to solve the problem. All you need to do is to add proper splitting tags to the code text inside that file and run the `%splitCodeForPackage()` macro.

Splitting tags behave similar to the help notes tags. When the framework encounters an opening tag, it evaluates to which destination it should redirect the code, and does it with every subsequent line until the closing tag is found. To learn more about this feature read the "*Developer's laziness*" section of the extended version of [**Jablonski 2020**]. A dedicated section in that article describes the structure of splitting tags, rules of code redirection, and provides some examples on this case.

[5] This is not exactly precise. The macro compilation does not care if the format and function exists when macro is compiled, it is only important that they will be there when the macro is executed.

HOW DO PACKAGES HANDLE DEPENDENCIES ON OTHER PACKAGES?

Since one of the goals of SAS packages is to support code reusability, you may expect that some developers do not want to reinvent the wheel and they can support their packages on other developers' work. This implies that there can be dependencies among packages. When a developer wants to indicate that a package requires another package to work, the developer adds the `ReqPackages:` tag to the `description.sas` file, and lists names and versions of required package(s). The list has to be a quoted, comma-separated list of the form: `"packageA(1.2.3)", "packageB(2.3.4)", "packageC(3.4.5)"`.

During the loading process, when the SAS Packages Framework sees that a package "X" is dependent on another package "Y," the framework verifies if the required version of the dependency package "Y" is already loaded. The SPF does it by inspecting the `&SYSLOADEDPACKAGES.` macro variable. If the package "Y" is not yet loaded (not in `&SYSLOADEDPACKAGES.`) the SPF starts recursive loading of "Y." If the loading of the "Y" package is successful then the SPF continues with loading package "X."

WHAT QUALITY STANDARDS DOES A PACKAGE NEED TO MEET TO BE ACCEPTED INTO SAS PACKAGES ARCHIVE?

For starters, you can visit the SAS Packages Archive location and find a document with SASPAC policy and "rules of engagement" for package contributors. The document lists general rules and quality requirements:

`https://github.com/SASPAC/.github/blob/main/SASPAC_policy.md`

Before a package is accepted to SASPAC its code can be subjected to a review. The bare minimum of code quality standards are the following:

- package introduces additional and non-trivial value to the community,
- package does not do intentional harm to users and users' environment (not to be mixed with improper use),
- package loads without issues or errors,
- every component of the package is documented (markdown documentation is a plus),
- package is tested and test documentation is provided,
- code properly handles exceptions, e.g., missing input data or improper parameter values,
- if any options are changed, they should be restored to the original values after execution,
- package is "legal", e.g., does not break intellectual property laws,
- package does not reveal illegal data, e.g., GDPR rules are not broken,
- package components are organized in line with type rules (i.e., macros in `macros`, functions in `functions`, etc.),
- package code is readable, in the sense that when source code is previewed, it is not a blob of letters, but a human being can read it (not necessarily understand, just read).

CAN MULTIPLE PEOPLE MAINTAIN A PACKAGE TOGETHER?

The answer to this one is short: Yes, absolutely! The collaboration details are of course left for developers, but in a package's `description.sas` file there are dedicated tags for: `Author:` and `Maintainer:`. The first one expects an author name, or a list of authors; the latter expects a maintainer name, or a list of maintainers. The difference between those two is: the author(s) is the person who came up with the idea (and potentially was the first maintainer); the maintainer(s) is the person who is currently responsible for the package development, maintenance, error fixing, and is a "first point of contact" for package-related cases.

WHAT IS THE DIFFERENCE BETWEEN A PACKAGE VERSION AND A BUNDLE SNAPSHOT, AND WHEN SHOULD I USE EACH?

In brief, package version and a bundle snapshot are independent and not related. Package version number is the indicator of the package release/life-cycle iteration. A bundle of packages is a file created from selected packages located in the `packages` fileref at a given point of time. The purpose of the bundle is to be a

snapshot of your packages "environment" saved for the future use, e.g. for audit trace purpose or execution of some historical analysis with setup from the "epoch." At a given moment, you can use three package with versions "X", "Y", and "Z", and have saved a bundle that contains those three packages with their historical versions from a year before, and no one will complain.

IS THERE A WAY TO CHECK WHICH VERSION OF THE SPF ITSELF IS INSTALLED?

Yes, there is. When you run any of the SAS Packages Framework macros with the `HELP` value as the argument, a help note dedicated to the macro is displayed in the `LOG`. In that help note, you will find information about SPF version.

CAN TWO PACKAGES DEFINE A MACRO OR FCMP FUNCTION WITH THE SAME NAME, AND WHAT HAPPENS IF THEY DO?

Yes, two packages can define a macro or FCMP function with the same name because each package is a separate entity. In fact, this property is not only limited to macros or FCMP functions.

The behavior of loading, when two packages define a macro or FCMP function with the same name, depends on which of those two types of code is loaded. For FCMP functions, the SPF loads them to separate data sets, named `<packagename>FCMP`. According to the `PROC FCMP` rules, the one that is loaded later is the one to be used; SAS will also issue a note in the `LOG` about it. Since compiled macros are located in the `work.sasmacr` catalog, also the one loaded later will be the one used. In case of macros, the second one will overwrite the first. SAS itself does this overwrite silently, but the framework checks if a macro with a given name already exists in the `work.sasmacr` catalog, and if one does, the SPF writes a note in the `LOG`.

CAN I ROLL BACK A PACKAGE TO A PREVIOUS VERSION AFTER INSTALLING A NEWER ONE?

By default, the SAS Packages Framework is designed to overwrite existing package files when the installation process is executed. To secure yourself from "losing" older versions you can do a few things:

- You can create a SAS packages bundle (using the `%bundlePackages()` macro) that can contain multiple packages archived together.
- You can move the old version to a separate location (using the `%relocatePackage()` macro).
- You can run installation macro the following way: `%installPackage(..., backup=1)` and create "timestamped" backup copy of existing file (works with `SPFinit` too).
- You can suspend automatic overwrite by running: `%installPackage(..., replace=0)`.

Eventually, if you did not do a copy, but the package is hosted in SASPAC, you can install the older version.

WHAT IS THE RELATION BETWEEN SAS PACKAGES AND R PACKAGES, IF ANY?

On multiple occasions I openly admitted that R packages were direct inspiration to the SAS packages idea and they influenced them greatly. The R environment allows its users to share their ideas, inventions, and code in an easy, almost seamless way. The vast ecosystems that have grown around R is, in my opinion, a competitive advantage that R has against SAS. Why did SAS, with its profound and historically well-established impact on data analysis, not embrace such a marvelous idea? I do not know... but I am convinced that sharing through packages can be a good thing for SAS.

CONCLUSION

In the article, I have answered a bunch of frequently asked questions that arose around the idea of sharing SAS code through SAS packages. I have shared references for further reading to expand your subject-matter knowledge. You have learned fundamentals of using SAS packages in the SAS session. The list of questions is not exhaustive and there is a fair chance it will grow in the future. Should you have other questions about the SAS packages idea, do not hesitate to ask them! I am looking forward to your input!

Answering to the question stated in the abstract, i.e., *Why has not SAS widely adopted the idea of sharing code through SAS packages?*, is non-trivial. It touches on multiple aspects, e.g., historical ways of code sharing, after all SAS is with us more than 50 years, customs and habits in the SAS programmers' community, or internal SAS Institute[6] policy and focus on different subjects. Maybe SAS packages is one of those initiatives where the SAS community has to be the driver of the change. That is why I urge and encourage you to engage! Play with existing packages; build your own; share them with your peers to build foundation for a strong and vital packages-based culture. See the motto on the SAS Packages Framework logo (Appendix B - SAS Packages Framework logo) and go with the motto: *SAS Packages - the way to share*!

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 is located at: https://github.com/yabwon/SAS_PACKAGES/blob/main/SPF/Documentation
and also: https://pages.mini.pw.edu.pl/~jablonskib/SASpublic/Paper_4725-2020_extended.pdf

[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 Troy Martin Hughes and Ryo Nakaya for their contribution in proof-reading and polishing the final article.

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.

[6] There are a few souls within the Institute that are supporting the idea and I am extremely grateful to them!

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.
```

Appendix B - SAS Packages Framework logo

Here is the SAS Packages Framework logo:

Appendix C - example of the `description.sas` file

Here is an example of how the `description.sas` file could look like, orange tags are obligatory, dark blue are optional:

code: example of the description.sas file

```
/* **HEADER** */
Type: Package
Package: mySASpackage
Title: This is a one-sentence description of my package.
Version: 0.0.1
Author: Bart Jablonski
Maintainer: Bart Jablonski
License: MIT
Encoding: UTF8

Required: "Base SAS Software", "SAS/ACCESS to SomeDataBase"
ReqPackages: "SQLinDS(2.3.3)", "basePlus(3.1.4)"

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

The **mySASpackage** package is an implementation
of my new idea how to do tome things easier in SAS

It is build for fun, but also (or foremost) to show
how easy it is to build SAS packages.

After all the motto is:
*"SAS Packages - the way to share"*

---
DESCRIPTION END:
```

INDEX