Paper OS11

Aspirations of language agnosticism - experiences expanding the programming stack with python

Kaj Arai Hejlesen, Novo Nordisk A/S, Copenhagen, Denmark
Michael Svingel, Novo Nordisk A/S, Herning, Denmark

ABSTRACT

The use of R within Novo Nordisk has expanded rapidly in recent years, with approximately 50% of studies now using R for some programming and analyses tasks, and an increasing number using R exclusively. With the expansion came an aspiration of language agnosticism: the ability to one day choose and use any programming language based on the task and/or competencies of the programmer(s) involved.

We present our experiences expanding our R-based GxP programming environment to include python for TFL programming and demonstrate that it is not just about installing the language binaries alone. Special consideration must be given to the processes & responsibilities associated with the environment as a whole such as package development, cross-language package sharing, systems validation and user support.

We conclude that mastery over these elements is fundamental to language agnosticism and indeed enables a more robust assessment of whether language agnosticism is a worthy aspiration at all.

INTRODUCTION

The landscape of statistical computing in clinical research has evolved significantly over the past decade. SAS has long been the gold standard for pharmaceutical companies due to its robust validation procedures, comprehensive documentation, and regulatory acceptance, while R has gained substantial traction in academic and clinical research settings owing to its open-source nature, extensive statistical libraries, and vibrant community support leading to cross-industry initiatives such as Pharmaverse.

However, the increasing complexity of modern clinical trials, including data visualization, and integration with diverse data modalities, has created new challenges that may benefit from expanding the traditional statistical computing toolkit. Python, originally developed as a general-purpose programming language, has emerged as a powerful contender in the data science and statistical analysis domain, offering unique advantages that complement the existing capabilities of SAS and R, especially within data domains such as image analysis and proteomics/genomics.

The integration of python alongside SAS and R in clinical research environments presents an opportunity to leverage the unique strengths of each platform while addressing their individual limitations. This multi-language approach can potentially enhance statistical analysis efficiency, improve code maintainability, facilitate advanced visualization and reporting, and provide greater flexibility in handling diverse data formats and sources commonly encountered in modern clinical studies.

This paper explores the practical implementation of python as a complementary tool to SAS and R in statistical programming workflows, examining the technical considerations, validation requirements, and best practices for incorporating python into a GxP compliant statistical analysis environment. We present a framework for leveraging python's capabilities while maintaining the rigor and reliability expected in clinical programming.

CREATING A GXP DELIVERY IN PYTHON

Creating a GxP-compliant delivery using python requires a structured approach that addresses both the computational environment and the software packages used within it. Python's flexibility and extensive ecosystem make it an attractive choice for regulatory work, but this same openness demands careful validation planning to ensure compliance with regulatory expectations.

The regulatory framework for software validation in pharmaceutical contexts stems from the FDA's 21 CFR Part 11 [1], which governs electronic records and signatures but primarily applies to transactional data collection systems rather than statistical analysis tools. The FDA's 2015 Statistical Software Clarifying Statement confirms that no

specific software is mandated for analyses, though version documentation is required for submissions. While 21 CFR Part 11 does not directly regulate statistical software itself, it's principles inform validation approaches when such tools operate within GxP-regulated systems, establishing expectations for reliability, traceability, and documentation.

A successful GxP delivery in python must demonstrate that results are accurate, reproducible, and traceable throughout the analysis lifecycle. This involves establishing a validated infrastructure, implementing appropriate controls for package management, and documenting all components used in the analytical workflow. The validation strategy should be risk-based, focusing resources on components that pose the greatest risk to data integrity and regulatory compliance. Unlike proprietary statistical software with vendor-supplied validation documentation, python's open-source nature requires organizations to take ownership of the validation process.

Given these regulatory expectations and the unique challenges of open-source software, this section outlines the foundational concepts and overall approach for building a validated python environment suitable for regulatory submissions. We distinguish between two critical validation domains: the compute environment and the python packages that execute analytical code. Understanding this distinction is essential for developing an efficient validation strategy that meets regulatory requirements without imposing unnecessary burden. The following subsections provide guidance on validating each of these domains within the context of GxP-regulated work.

VALIDATION OF COMPUTE ENVIRONMENT

Establishing a compliant compute environment for Good Clinical Practice (GcP) and broader GxP activities requires a systematic approach that balances regulatory requirements with practical usability. The compute environment forms the foundational infrastructure upon which python packages operate and analytical code executes, encompassing the operating system, python interpreter, system libraries, dependency management tools, and runtime configurations. The purpose of system qualification is to ensure reproducibility of expected results across the analytical environment, not to reassess the risk of individual packages.

The infrastructure validation is managed centrally by a dedicated team responsible for maintaining a standardized, pre-qualified computing environment. For this GxP delivery, the compute environment builds upon a golden image provided and maintained by the centralized standards team, which has been qualified through formal IQ/OQ/PQ protocols including verification of system specifications, security configurations, and baseline functionality. While the golden image provides a validated foundation, project-specific validation activities remain necessary to demonstrate that the environment supports the particular analytical requirements of this delivery. This is achieved through the use of Docker containers that encapsulate the analytical environment, providing isolation, reproducibility, and version control for all software components beyond the base infrastructure. The Docker container specification (Dockerfile) serves as executable documentation, explicitly defining all installed packages, their versions, and installation parameters, ensuring reproducibility across different execution environments.

Functional specifications have been established using Gherkin syntax, providing human-readable, structured requirements that define expected environment behaviors in a Given-When-Then format. These Gherkin specifications are directly implemented as automated pytest test cases, creating full traceability from requirements to test execution and establishing a living documentation system that validates environment functionality. The automated test suite verifies critical environment capabilities including: successful package imports, correct installation of all specified components, absence of dependency conflicts, selective execution of package-level test suites that reflect typical usage patterns, proper interaction between packages and system libraries, and validation that analytical functions produce expected results within the containerized environment. Where packages include their own test suites, representative tests are incorporated into the qualification protocol, with test selection based on relevance to the analytical workflows employed in this delivery.

Each test execution generates documented evidence of operational qualification, with pass/fail results traceable back to specific functional requirements defined in the Gherkin specifications. This requirements-based testing approach aligns with regulatory expectations for software validation, providing clear evidence that the environment has been tested against predetermined specifications and quality attributes to ensure consistent, reproducible performance.

VALIDATION OF PACKAGES

According to the FDA's Glossary of Computer System Software Development Terminology, validation is defined as "establishing documented evidence which provides a high degree of assurance (accuracy) that a specific process consistently (reproducibility) produces a product meeting its predetermined specifications (traceability) and quality attributes." [2]

This section outlines how to ensure the accuracy of python packages when used as part of a validated environment. Since python is open source, ensuring that the environment is reproducible and traceable presents challenges that must be addressed through appropriate controls and documentation.

Accuracy:

Python packages can be differentiated by their development characteristics:

- Standard library packages - maintained by the Python Software Foundation through a rigorous development lifecycle including formal governance (PEPs), comprehensive test suites, strict version control, and public issue tracking. There is minimal risk in using standard library packages as components in a validated system.

- Third-party packages - developed by individuals or organizations with varying levels of development rigor. While widely-used scientific packages (NumPy, SciPy, scikit-learn) often maintain high development standards including extensive testing and peer review, quality is not guaranteed.

Unlike R, where core statistical functionality is included in base packages, python's standard library contains minimal statistical or analytical capabilities. The majority of packages used for data analysis, statistical computing, and scientific workflows are third-party packages. Therefore, for python-based GxP systems, nearly all packages used in analytical workflows require accuracy assessment. A risk assessment should be conducted to evaluate the accuracy and validity of each third-party package with respect to its intended use.

Reproducibility:

Python's continuously evolving package ecosystem presents reproducibility challenges. Package dependency trees can be large and complex, with dependencies on specific python versions, system libraries, and external software. Ensuring compatibility requires explicit dependency management through tools such as virtual environments, dependency lock files, and containerization.

Traceability:

Traceability requires documentation of all packages used in an analysis. A risk-based approach may focus validation on packages directly used in analytical code while documenting their dependencies. This should be supported by standard operating procedures, code review processes, and automated tools that capture the complete package environment within logged workflows.

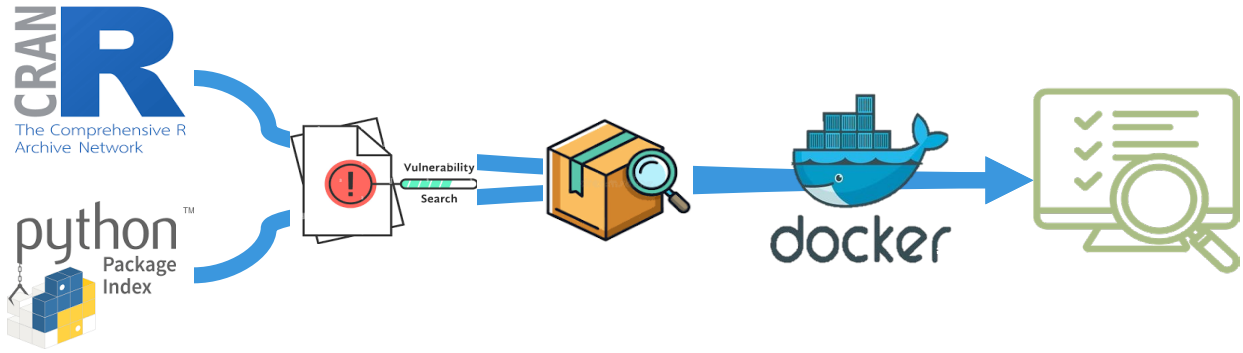

Figure 1: approval workflow of packages

VALIDATION OF PYTHON PACKAGES IN PRACTICE

The validation of python packages in this study follows a structured, risk-based approach consisting of three primary steps:

1. Package Request

When a python package is required for use in the validated environment, a formal request must be submitted. This request includes:

- The package name
- A clear description of the use case and analytical purpose
- Justification for why the package is necessary

2. Risk Assessment

Each requested package undergoes a comprehensive risk assessment based on four criteria:

Purpose:

The intended use of the package and whether it performs statistical computations. Statistical packages (those implementing statistical algorithms, machine learning methods, or mathematical modeling functions) inherently present greater risk as errors in complex algorithms may be difficult to detect and can directly impact study conclusions. However, all packages undergo the same validation testing regardless of their classification.

Maintenance Good Practice:

The package is evaluated for evidence of sound software development practices. Metrics include:

- Presence of comprehensive documentation and vignettes
- Public source code repository (e.g., GitHub) with visible development activity
- Formal issue tracking and bug reporting mechanisms
- Release history and version management practices
- Size and complexity of the codebase
- Type of license and governance model

Community Usage:

The level of exposure to the user community serves as a proxy for real-world testing. Metrics include:

- Package maturity and time since initial/latest release
- Number of downloads and active users
- Number of reverse dependencies (other packages that depend on it)
- Consideration of trusted resources: Packages sponsored by NumFOCUS (NumPy, SciPy, pandas, scikit-learn) or maintained by the python Software Foundation may be considered lower risk due to established governance and development standards.

Testing:

The presence and quality of the package's test suite. Metrics include:

- Existence of unit tests within a standard testing framework (pytest, unittest)
- Code coverage percentage
- Continuous integration practices (automated testing on commits)
- Test documentation and traceability

The risk assessment is documented in an individual package report that assigns a risk level (High, Medium, or Low) based on the evaluation of these four criteria. The assessment should be performed by a qualified individual with appropriate statistical programming experience and reviewed by a similarly qualified colleague.

3. Package Installation and Validation

Approved packages are installed in the validated computational environment with explicit version pinning. The installation and validation process includes:

Installation:

- Installation from a controlled repository (internal mirror behind a vulnerability search engine)
- Documentation of the exact package version and all dependency versions
- Creation of a locked environment specification (e.g., requirements.txt with pinned versions)
- Version control of the environment specification

Import Verification:

Confirmation that the package can be successfully imported without errors.

Unit Test Execution:
The package's existing unit test suite is executed within the validated environment. This serves as regression testing to ensure the package behaves as expected in the specific computational environment. Tests are allowed to fail if they are demonstrably irrelevant to the intended use case (e.g., tests for optional dependencies not installed, platform-specific tests for unsupported platforms). Any test failures must be documented and justified.

Documentation:
All validation activities, test results, and approval decisions are documented in the package validation report. This report serves as documented evidence of package accuracy for regulatory purposes.

The complete validation workflow ensures that python packages used in GxP-regulated analyses meet requirements for accuracy, reproducibility, and traceability. The risk-based approach focuses the risk assessment on requested packages while maintaining consistent validation testing across all packages in the computational environment.

WORKING MULTILINGUAL WITH R AND PYTHON

In Novo Nordisk we work with two different compute environments. On one hand, SAS Viya offers the SAS language, on the other, our POSIT platform has several offerings, including R and python. This provides different options for both the data formats as well as the code execution, as both language offers the option of executing code in the other through function calls, handling the conversion of inputs and outputs implicitly.

Currently there currently plenty of resources when working with R in clinical development, both internally in Novo Nordisk and externally in the Pharmaverse and others, do we then need to maintain a complete library in both R and python. Afterall, we could just use python for the use cases where it is needed, while relying on R packages for anything else, including handling data imports and the final formatting of the TFL.

For this experiment, we chose to utilize our existing R packages for importing and exporting data as well as handling the formatting of the TFL. This was done to test if this approach would be viable and therefore potentially would mean we did not have to maintain similar functionality in two languages. Finally we used the open-source package Whirl for handling the execution of the code, creating the logs with relevant information. While this is an R package, it is designed to handle code from any language.

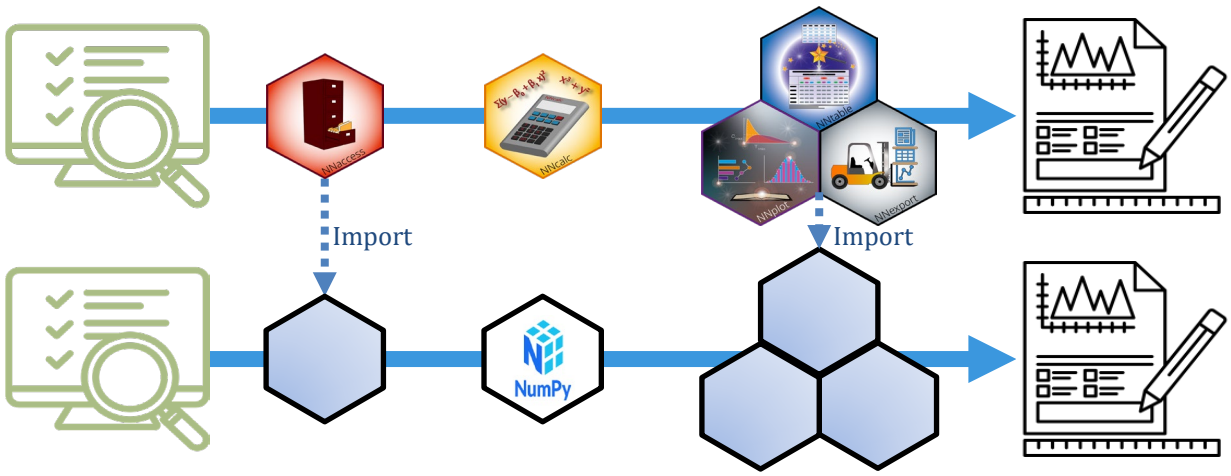

Figure 2: Using rpy2 to import R framework for import/export and formatting

RPY2

rpy2 is a python library that enables seamless integration of R functionality directly into python scripts by embedding the R runtime within python. This bridge allows python users to call R functions, access R packages (including CRAN and Bioconductor), and leverage R's extensive ecosystem without leaving the python environment. For our project, rpy2 was the key technology enabling us to maintain python as the primary analysis language while accessing established R packages for data handling and output formatting.

rpy2 operates by creating a bidirectional interface between python and R, with conversion of objects between the two languages. The most straightforward approach is to work directly with R objects:

```
#!/usr/bin/python3
import rpy2.robjects as robjects
from rpy2.robjects.packages import importr

# Import base package from R into python session
base = importr('base')

# Create R vector using robjects
r_values = robjects.FloatVector([1, 2, 3])

# Call R's mean function
mean_value = base.mean(r_values)
```

In practice, analytical workflows often involve python objects such as numpy arrays or pandas DataFrames. rpy2 can convert these to R objects, but requires explicit conversion context:

```
#!/usr/bin/python3
import numpy as np
import rpy2.robjects as robjects
from rpy2.robjects.packages import importr
from rpy2.robjects import numpy2ri
from rpy2.robjects.conversion import localconverter

# Import base package from R into python session
base = importr('base')

# Create numpy array (typical in python workflows)
values = np.array([1, 2, 3])

# Use conversion context to allow rpy2 to convert numpy to R
with localconverter(robjects.default_converter + numpy2ri.converter):
    mean_value = base.mean(values)
```

The localconverter context manager enables automatic conversion between python and R data structures within its scope. Similar converters exist for pandas DataFrames (pandas2ri.converter), which was particularly useful in our project where data manipulation was performed in pandas before being passed to R for formatting.

DATA TYPE CONVERSION ISSUES

As discussed earlier, we used our internal R package for data import to leverage its validated infrastructure and organizational standards. One of the most significant challenges we encountered was data type conversion when reading SAS datasets through R into python. This multi-step conversion process (SAS → R → python) introduced unexpected type transformations that required careful handling.

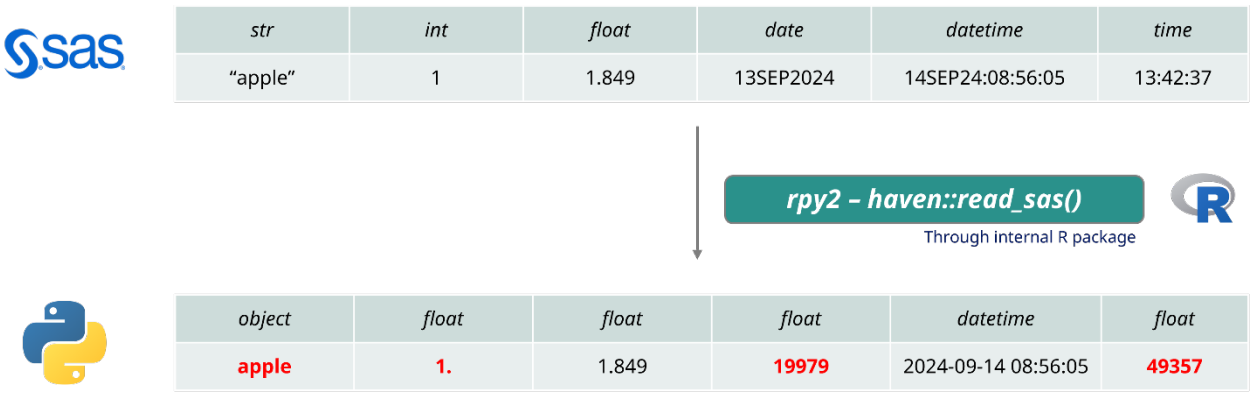

Figure 3: Conversion issues we found

Figure 3 illustrates the data type transformations that occur when reading a SAS dataset through rpy2's haven::read_sas() function. The original SAS data types (character strings, integers, floats, dates, datetimes, and times) undergo systematic conversion as they pass through the R layer into python. Most notably, SAS character variables become python object dtype rather than string, SAS integers convert to python float64, and dates and times are reduced to their underlying numeric representations as floats. For example, the date "13SEP2024" becomes the float value 19979 (days since the SAS epoch of January 1, 1960), and the time "13:42:37" becomes 49357 (seconds since midnight). This loss of type information means that python receives no indication that these float values represent temporal data, requiring manual reconstruction of the original semantic meaning.

The root cause of these conversion issues lies in the fundamental differences in how each system represents data. SAS stores all numeric data as floating-point numbers with associated format metadata that determines display and interpretation. When R's haven package reads SAS files, it attempts to preserve some of this information through R's class system (such as converting SAS dates to R's Date class), but rpy2 conversion layer then maps these R objects to the closest python equivalents, often defaulting to generic types like float64 and object. This multi-step translation process, while functional, demonstrates a broader challenge in data interoperability: each language has evolved its own conventions for representing common data types, and bridging between them requires either lossy conversions or complex metadata preservation systems.

STANDARDISATION, STANDARDISATION, STANDARDISATION

While working with a multi-lingual project like the one described in this paper, the importance of standardisation quickly becomes apparent. Several of the issues we faced concerned lack of proper data standards, using language specific data formats, making implicit conversions resulting in mismatch of data types and handling of missing data.

In Novo Nordisk we are in the process of moving to a new statistical computing environment, which will enable us to apply standards to a higher degree than present. The learnings from this experiment will attribute to our design choices, to be able to support the expansion of language options.

DATA STORAGE AND DATA TYPES

The industry standard for file storage has for a long time been limited to a file-based system, typically using the proprietary sas7bdat format to store data. This has been driven by the exclusive use of SAS as analytical language and regulatory requirements such as submissions using SAS XPORT data format. In later years as R has become increasingly used, the R-based data format RDS has become more prominent. However, both suffers from not being truly language agnostics, but storing data in a format suitable for a single language.

In our experiment we suffered from several issues owing to data conversions. Floating points numbers are defined differently; missing values can be represented as different NA or NULL values and time formats can be confused with datetime formats or be off due to different definition of starting date (1960 vs. 1970).

We propose it as a necessity to develop a data storage with data stored in a format not tied to a specific language, but either in a database or an open-source format. If using a file-based storage options could be formats like parquet or dataset-json. In Novo Nordisk we are moving from a file-based system with data primarily stored as sas7bdat to a cloud based system with Databricks as the data repository, storing data in their delta table format, with parquet in the backend. We still plan to allow for file-based storage, but using parquet files instead of formats like sas7bdat or rds.

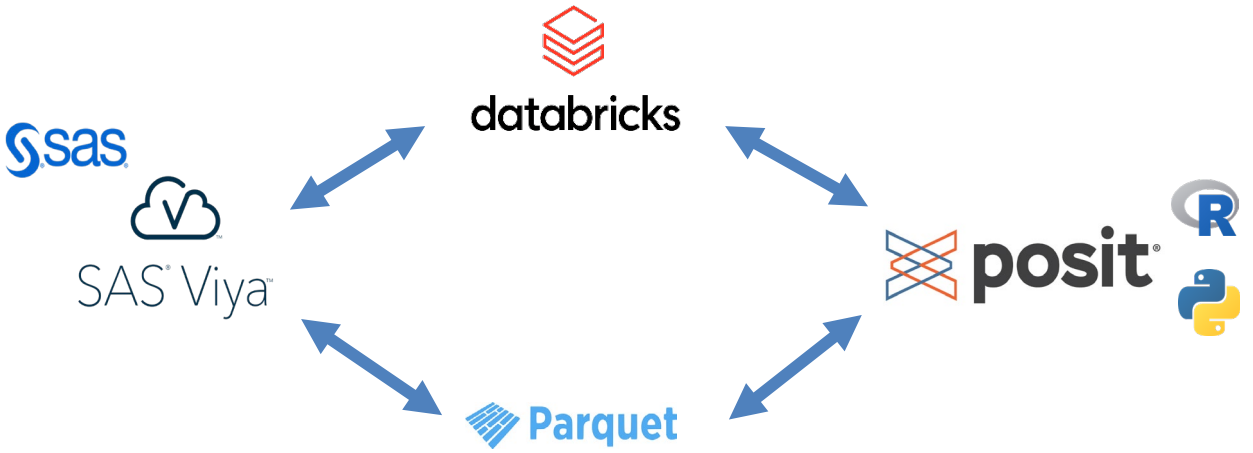

Figure 4: High level architecture of a cloud-based system, still allowing for file-based storage

In this cloud based system, we will have two different compute environments, with a total of three primary language options. To avoid converting between the native language formats, it is important to develop tools to import and export data for each language, not just for each compute. The standardization occurs at the database level, and it is the responsibility of each language to be able to export the data in the expected format.

USING STANDARD SPECIFICATIONS

While certain language-dependant tools may need to be created and maintained, the *specifications* controlling them should be written in a standard language-independent format, such as YAML or JSON. Like the data repositories, this should be stored in a location accessible from all platforms, for instance through an API or a central repository.

- Using **language-independent** formats like YAML or JSON for all specifications
- Using **language-dependant** readers to translate specifications into jobs
- Stored in a **central repository**, accessible through eg. API or a central (GIT) repository.

In Novo Nordisk we employ both strategies. For the end-to-end specifications detailing the specifications and mappings of the SDTM framework and specifications of the metadata to ADaM and TFL generations we use OpenStudyBuilder, which exposes the specifications through an API. For more specific specifications, we store these as YAML file and version this in a central GIT repository together with the study code. This way, we have separated what specifies the data flow to what executes it, all the way from SDTM generation to TFL.

CONCLUSION

SAS has long been the only programming language accepted in the pharma industry for the analyses of clinical data. Recently the open source language R has emerged as an alternative, offering more options. To date several companies have created submissions partially or fully in R, while in Novo Nordisk around one third of all programming is now being done in R. As studies is becoming more complex with more data types, even R might no longer be enough. In this paper we explored the possibility of adding an additional language, python, to the line of offerings to meet these needs.

We created an experiment using python for a specific delivery, a periodic safety update. We demonstrated it was possible to deliver a TFL created in a GxP-compliant python environment, using the same process as the validation of R and associated packages. To limit the effort on maintaining complete frameworks in different languages, we explored the use of the rpy2 package, thereby utilizing existing R packages for import/exports as well as formatting of outputs. While possible, it also presented challenges, mainly related to conversion errors between the different data formats.

We conclude that it is possible to incorporate python or any other language into a GxP-validated framework, but by doing so being mindful of the challenges that comes with it. We recommend the focus to be on standardising, from the data formats, through specifications and the tools used. All these should exist separate of any one particular language, while the functionality to interact with them should be maintained from each language, to truly have a language-agnostic setup.

REFERENCES

1. U.S. Food and Drug Administration. (2003, August). Guidance for industry: Part 11, electronic records; electronic signatures — scope and application. U.S. Department of Health and Human Services. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/part-11-electronic-records-electronic-signatures-scope-and-application
2. Nicholls, A., Bargo, P. R., & Sims, J. (2020, January 23). Pharmaverse white paper: A new paradigm for collaboration in pharma. R Validation Hub, R Consortium. https://pharmar.org/white-paper/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Author Name: Kaj Arai Hejlesen
Company: Novo Nordisk A/S
Address:Østmarken 3A, 2860 Søborg, Denmark
Work Phone: +45 3079 0515
Email: kjhl@novonordisk.com

Author Name: Michael Svingel Wass
Company: Novo Nordisk A/S
Address: Østmarken 3A, 2860 Søborg, Denmark
Work Phone:
Email: wymz@novonordisk.com