VisGap'23 – The Gap between Visualization Research and Visualization Software (2023)
C. Gillmann, M. Krone, G. Reina, T. Wischgoll (Editors)

Better Information Visualization Software Through Packages for Data Science Ecosystems

R. Henkin[1]

[1]Queen Mary University of London

Abstract

Good software development practices are important factors for the successful translation of visualization research into software. This paper argues for the creation of packages for data science ecosystems, with Python and R as case studies, as a way to employ existing tools and infrastructure towards better information visualization software. The paper describes open practices, sustainability and FAIR software to motivate package development. The ecosystems of Python and R are then reviewed based on general software development aspects and how common features of visualization software, such as rendering and interactivity, are supported. It concludes with the software engineering benefits related to creating packages in Python and R and initiatives to overcome obstacles that may hinder the development of better software.

CCS Concepts

• ***Software and its engineering*** → *Software creation and management;* • ***Human-centered computing*** → *Visualization; Human computer interaction (HCI);*

1. Introduction

Many issues that prevent the successful translation of visualization research into software overlap with the general theme of software sustainability, which includes development practices to ensure software continues to work in the future, the adaptation of existing software to address new needs, training of people to perform these tasks, and the funding that will support all of the former. Some of these aspects overlap with open practices [JKA*17] and are also part of the drive towards open and FAIR (Findable, Accessible, Interoperable and Reusable) software in research [HCH*20], which aims to increase transparency, reproducibility and reusability of research, as an extension of FAIR principles for scientific data [WDA*16]. Achieving all of this requires not only practice changes in software development but also active involvement by all stakeholders in research, from programmers to project supervisors.

In the visualization community, some authors have discussed open practices [Har18] and better software development [RCM*20], but not every openness demand or development suggestion is immediately actionable by researchers creating software, nor do they necessarily lead to better software in isolation. In this paper, I encourage the visualization research community to use package creation in data science ecosystems as a more accessible route towards better software, using the Python and R programming languages as case studies.

These two languages lean heavily on a system of open-source package distribution that empowers data-related work, with ecosystems that can motivate the creation of FAIR software. They also provide great opportunities for visualization researchers to deploy software that can reach large, multidisciplinary audiences and build on and contribute with tried-and-tested software. Tools in Python and R ecosystems and existing initiatives can also contribute to creating more open and sustainable software.

As packages, exploratory interfaces or visual analytics systems can be deployed in their complete form or broken down into individual components. Once published, they can be reused again by visualization researchers or directly support data analysis activities by end users. Indeed, projects from the visualization research community have previously deployed software as packages for these languages [MQB19, OCL*21, NSS21], while others have been successfully integrated *a posteriori* into the ecosystems, ranging from APIs for visualization grammars such as Altair [VGH*18] and new plots such as UpSet [CLG17].

However, these all are separate examples with their own incentives and motivation that depended on the context of those projects. Besides examining the technical aspects of maintaining and testing packages, this paper also explores community-level initiatives, centered on packages created as open-source software, as attempts to overcome systemic problems of incentives and motivation.

The rest of the paper is structured as follows: in section 2, I introduce accepted definitions of open practices, sustainability and FAIR principles and describe existing initiatives for improving software in academia; in section 3, I characterize the Python and R ecosystems and contextualize them in relation to sustainable and FAIR software; in section 4, I describe how visualization software

DOI: 10.2312/visgap.20231115

can be developed in Python and R: how and which desired features of visual tools are currently supported if researchers want to create visualization packages; finally, in section 5, I discuss how existing tools and infrastructure and open-source community initiatives help to facilitate sustainability and FAIR software through package development, as well as obstacles that may hinder it.

2. Open, sustainable and FAIR software

Translating research into good software relates to many different issues as mentioned in the introduction. This section briefly describes three topics — open practices, sustainability and FAIR software — that are interconnected and contribute with motivation and ideas to inform better visualization software in the form of packages.

2.1. Open practices

Open practices have a long tradition in software development in the form of open-source software, where its source code is made available and accessible for modification, derivation and redistribution. In addition to demands from funders, there are many different justifications for making code public and open; in research, concerns about reproducibility have driven the call for openness [IHG12, SMB*16], as well as the value of openness for better software [Bar10]. In visualization research, openness has until now been discussed more in the context of experimental results than software [Har18] and as a conversation *within* the community. However, the same concerns and benefits that are advocated for science in general *should* also be valid for visualization research, and the lack of scientific discussion regarding open practices does not mean that the practices do not happen. For example, in a quick search for *GitHub*, *OSF.IO*, *Zenodo* and *Figshare* (which are all commonly used repositories for open data/software) in abstracts published in key visualization journals (IEEE TVCG, Computer Graphics Forum and Information Visualization), the number of published articles found tripled from 2020 to 2022, from 12 to 43. The objective of this paper, however, is to encourage researchers to go beyond copying files to open repositories and actually transform prototypes into well-designed software packages, while also acknowledging that the visualization research community includes a diversity of backgrounds and that some researchers may have limited programming experience.

2.2. Sustainable software and research software engineering

Sustainable software relates to ensuring that software created from research continues to work and is improved or adapted beyond the end of a project. Challenges to sustainable software involve motivation, funding, personnel and infrastructure, among others [ABD*21]. Enabling the creation of sustainable software, therefore, encompasses software development practices as well as initiatives that go beyond programming and become related to departments, institutions, countries and research communities. In the United Kingdom, for example, the Software Sustainability Institute was created in 2010 with the objective of "cultivating world-class research with software" [CHH*13]. The institute organizes events such as the "software carpentry" and collaboration workshops and also offers other sources for researchers aiming to improve their software development abilities. The institute was founded by one of the UK's research councils, with the recognition of the need for sustainable software but also with a demonstration of how these initiatives depend on local and global research environments.

At the same time, this institute has also driven the emergence of research software engineering (RSE), with an associated job that, as the name suggests, involves the creation of software for research through best practices of software engineering [CKB*21]. The concept and careers are now recognized worldwide, with associations in various countries and continents (`https://rsse.africa` and `https://rse-asia.github.io/RSE_Asia`). These associations promote the recognition of the role of software in research and many of their aims align with the ideas of software sustainability focused on the people involved with it.

The creation of packages for visualization software does not make software sustainable by definition, but, combined with open and better software engineering practices, may be able to *help* researchers follow the path towards better software. For example, in the countries and institutions where research software engineers are recognized, there may be a suitable role for them in research projects at the time of writing proposals.

2.3. FAIR software principles

Initiatives for open practices have come alongside the movement for better access to scientific data, in particular, the FAIR Guiding Principles [WDA*16], which established guidelines for making scientific data *findable, accessible, interoperable and reusable* from a machine perspective. In short: findable means that users and computers are able to find the data by using, for example, title, authors and keywords; accessible data provides methods and protocols that enable the computers to retrieve the data, using authentication procedures if needed; interoperable means that data representations are not unique for that dataset and incompatible with other datasets; finally, reusable data is published with enough information that enables others to use the data (e.g. the meaning of columns in tables) or attempt to replicate the data (e.g. which experiment was used to produce the dataset?). Although these principles are not equivalent to open data, they are motivated by the same need for scientists to access, reproduce and reuse previously published data, especially through the use of computational resources to facilitate managing large volumes of data.

More recently, researchers involved in software sustainability have extended the idea of FAIR data to FAIR research software, called FAIR4RS, with similar concerns regarding the reuse of software in research [HCH*20]. Although the motivation comes from fields where reproducible computational workflows are more important, applying FAIR principles in visualization could help to make visual analytics systems, which are more difficult to reproduce than algorithms or pseudo-code, more accessible and reproducible, for example. The FAIR4RS principles [BCHK*22], shown in table 2.3, relate to the discovery and reuse of software based on central repositories for human and machine discovery, as well as interoperability based on the use of common data formats. One important consideration of the FAIR principles is that they are guidelines rather than a full set of rules that developers must adhere to; developers must consider which principles suit their objectives and adapt as necessary. As it will be shown in the next few sections, package creation ticks many of these boxes.

F: Software, and its associated metadata, is easy for both humans and machines to find.
A: Software, and its metadata, is retrievable via standardised protocols.
I: Software interoperates with other software by exchanging data and/or metadata, and/or through interaction via application programming interfaces (APIs), described through standards.
R: Software is both usable (can be executed) and reusable (can be understood, modified, built upon, or incorporated into other software).

Table 1: *FAIR4RS Principles (first level). Adapted from [BCHK*22] which is licensed under the Creative Commons Attribution 4.0 International License* http://creativecommons.org/licenses/by/4.0/.

3. Python and R Ecosystems

The languages and ecosystems examined here — Python and R — are, at the time of writing, among the most popular for data-related work. They are not the only ones used, nor they are exclusively designed for data: whereas R has origins in statistical computing and graphics, Python is a general-purpose language. However, they share enough similarities that made them popular for data analysis and with strong support for data visualization, in particular information visualization. This section describes some of the most important characteristics of the ecosystems surrounding these languages that influence the creation of visualization software, without discussing how programming in those languages influences the software.

3.1. Software distribution

Doing data analysis or writing software in both Python and R is based on loading packages (or modules) with functions or variables that the standard set of functions (often included in "base" packages) does not support. For Python, this is mandatory for any programmer, as the Python Standard Library (included with every installation) does not contain functions directly related to data analysis (e.g. plotting). R, on the other hand, due to its statistical origin, includes many statistics-related functions as part of *any* R installation. Both ecosystems feature official software repositories from which users download and install packages, with their own rules for the submission and acceptance of new packages.

The R package repository is part of The Comprehensive R Archive Network (CRAN), which also hosts distributions of the R language itself. CRAN is managed by a team of volunteers and package submissions must pass an automatic check procedure (http://cran.r-project.org/web/packages/policies.html), with the first submission often needing a manual check. The procedure does not ensure that the package works as intended, but rather if the package is in a state that it can be included in the repository, that is, it can be compiled across multiple operating systems, the documentation passes the minimum requirements, among other tests.

Python packages are often distributed through the Python Package Index (PyPI) (https://pypi.org), managed by the Python Software Foundation (a non-profit corporation). Compared to CRAN, this repository has a minimal set of rules which a package must comply with, generally more related to package metadata than runnable code; indeed it is possible to publish packages that simply do not work. In the past decade, the Python ecosystem has also been heavily influenced by the emergence of Anaconda (https://www.anaconda.com), a corporation-managed Python distribution that by default includes many packages and is aimed at facilitating the setup of a programming environment (and which also supports R). The company also hosts its own repository, from where Anaconda users will *typically* install their packages — users can still use PyPI and the Anaconda repository side-by-side.

A final note on this is that it is possible to install packages without using these repositories, by sharing release files or setting up open-source repositories like GitHub. However, all the key repositories and their standards are compatible with the FAIR software principles, and reusing published software and data formats also ensures that more principles are covered.

3.2. Programming environments

There are multiple ways to analyze data or use interactive visualizations in Python and R: executing programs that create outputs, using Integrated Development Environments (IDEs) or literate programming (or computing) environments such as Jupyter Notebooks. IDEs used for data analysis usually have sub-windows separating code from visual outputs and encourage exploration and iteration. They are more limiting for interactive visualizations that have multiple views or that are integrated into pipelines where user interaction is used to connect different steps (e.g. selecting subsets of data). However, it is also possible to launch web apps that load data directly from an IDE after a user has preprocessed data, for example.

Literate environments support the creation of documents where code and outputs are mixed with text for documentation or to provide context in the form of narratives. Platforms and types of documents for literate programming vary from highly interactive documents like Jupyter Notebooks to more traditional code writing like R Markdown. The Jupyter platform supports Python, R and also other programming languages. Interactive visualizations used in these documents are web-based and often self-contained, meaning that, for example, it is difficult to select objects in one plot and reuse the selection in another plot on the same document. The Jupyter platform can be run as a server or on the user's computer, but some services allow publicly sharing collections of notebooks (https://mybinder.org). Notebooks can also be exported as stand-alone HTML or PDFs: in these cases, the packages must also have a method to export static graphics so that the visualizations can be included in those documents.

The platform also supports the creation of interactive *widgets* outside of the notebook, i.e. independent of the actual code written in the notebook. These widgets can be form-like controls (slider bars) and also interactive visualizations [ISLH19, MT20]. These are developed targeting the notebook platform itself and are not typical packages for either Python or R. However, due to the popularity of the platforms, researchers have been investigating data work practices and the role of interactive visualization there as well [SO20, GCL21, WHS20].

3.3. Deployment of user-facing applications

Python and R strongly support "workbench" analysis through interactive environments and visual interfaces, but they also support the deployment of applications and interfaces for remote access, for example, in the form of interactive web apps. The deployment methods vary between commercial and open-source, and language-specific and language-agnostic. For example, container methods such as Docker enable deploying applications that neither require users to be proficient in a programming language, nor ask them to install whole environments for any of the languages. There are also many commercial cloud-based options that support both Python and R applications.

For R web apps, which would very likely be developed using the *shiny* package (see 4.2), there is also the option of the open-source Shiny server by the Posit company (creators of the RStudio IDE) that can be self-hosted by any institution. The same company also offers the widely used shinyapps.io website, which has free and paid-for plans for hosting Shiny apps. For Python web apps, the options will depend on the framework used for development; popular frameworks such as Flask have several open-source and commercial options (`https://flask.palletsprojects.com`).

For both ecosystems, it is common to distribute web apps as packages; either on their own or complementing other pieces of code. For example, a visual analytics system can be distributed in the form of a single function that a user executes from a script or command-line interface and creates a local server, while an interactive visualization component can be distributed as a self-contained package alongside a web app that demonstrates its functionality and/or customization options.

3.4. Cross-language interoperability

Another important aspect of these languages for visualization research is interoperability, which includes, for example, the ability of code written in one language to interact with code written in another language. There are multiple reasons for its importance, from running more efficient methods that have been implemented in other languages to accessing state-of-the-art packages that are available in one language but not others. Between Python and R, language-level interoperability is normally done through packages such as *reticulate* in R [UAT22] and *rpy2* [Gau08] in Python. The reticulate package supports a variety of methods, including converting equivalent object types from Python to R and running Python inside an interactive R session, essentially mixing both languages in real time. The rpy2 package works similarly, supporting the creation of functions that are written directly in R, installing additional R packages from within a Python environment and accessing R objects. There are also other methods for interoperability across other languages, for example, at process level or using programming environments that support that (e.g. RStudio), but these are not exclusive features of these ecosystems for visualization software.

4. Support for features of visualization software

The previous section focused on the general factors surrounding the development of packages for Python and R. When adapting or developing outputs of visualization research as packages, there are some features of these outputs that researchers will want to preserve as much as possible in any context. This section describes some of these features relative to both Python and R ecosystems, including existing packages and programming environments. These features are described mostly from an information visualization perspective — scientific visualizations may have different requirements not covered here.

4.1. Visualization rendering and outputs

R, with the standard set of packages, supports drawing static plots to *graphic devices* such as windows in the computer, PDF, bitmaps or SVG files, among others. The standard R installation includes functions to draw basic 2D X-Y plots, but it is also possible to create a whole plotting engine or set of functions using "graphical primitives" such as lines, rectangles and other shapes. For 3D drawing, the *rgl* [MA22] package includes high-level operations for drawing plots and drawing shapes, as well as lower-level operations such as camera control and interaction. Python does not have an "inbuilt" package for rendering and relies purely on packages for drawing static plots and graphical primitives. For static plots, both ecosystems also have well-established packages such as *ggplot2* [Wic16] in R and *matplotlib* [Hun07] in Python; their popularity also led to the development of other packages to extend functionalities and include new types of plots. One of the limitations of these two packages is that the outputs are static: interactivity can only be added through complex coordination with graphical user interface (GUI) packages that enable tracking mouse position and clicks.

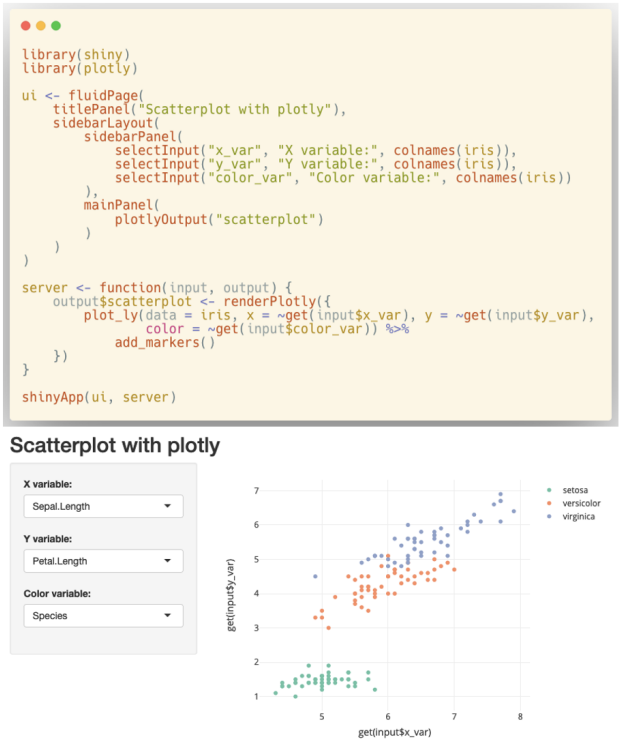

Figure 1: *Simple example of an interactive scatterplot based on Plotly and Shiny in R.*

```python
import streamlit as st
import pandas as pd
import plotly.express as px
from sklearn import datasets

# Load the iris data
iris = datasets.load_iris()
iris_data = pd.DataFrame(iris.data, columns=iris.feature_names)
iris_data['species'] = iris.target_names[iris.target]

# Create a plotly scatterplot
fig = px.scatter(iris_data, x='sepal length (cm)', y='sepal width (cm)',
color='species')

# Add a selectbox widget for the X axis
st.sidebar.markdown('Select the variable for the X axis:')
x_var = st.sidebar.selectbox('X axis', iris_data.columns)

# Add a selectbox widget for the Y axis
st.sidebar.markdown('Select the variable for the Y axis:')
y_var = st.sidebar.selectbox('Y axis', iris_data.columns)

# Update the scatterplot with the selected variables
fig.update_traces(x=iris_data[x_var], y=iris_data[y_var])

# Show the scatterplot
st.plotly_chart(fig)
```

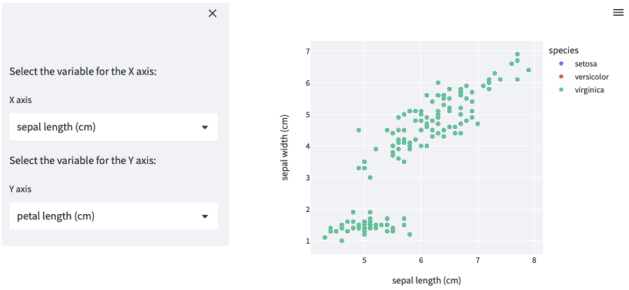

Figure 2: *Similar example of fig. 1 but written in Python with Plotly and Streamlit.*

Python and R developers have also created packages that produce web-based graphics and rely on modern JavaScript for interactivity. Some of the packages expand on existing information visualization libraries already implemented in JavaScript, such as *Plotly* (https://plotly.com), and *Altair* [VGH*18], which is built on top of *Vega* and *Vega-lite*. Others, such as *Bokeh* [Bok18] in Python, were developed from scratch, and led to the development of other packages such as *HoloViews* [RSB*22].

4.2. Coordinated and multiple views

Individual plots are the backbone of visual analytics systems or exploratory tools, but coordinated and multiple views (CMV) tie all the plots together in a user interface. Depending on the programming language of choice, visualization designers may have to spend programming hours on both the data visualization side and the construction of the interface. For JavaScript visualizations based on a library such as *D3.js* [BOH11], the charts often need to be integrated with web app frameworks such as *React*, introducing new learning obstacles or another layer of software dependency. In both Python and R, frameworks have been developed to enable developers to keep their focus on the backend data computations and the visualizations. Although they all introduce their own language constructs and changes in programming paradigms, developers should be able to create functioning prototypes more easily.

For web-based tools, the *shiny* package [CCA*22] (R, as of 2022 also available for Python) is a web application framework based on reactive programming that enables developers to use R functions to construct HTML interfaces and write code that links the visual outputs to the computations. A simple explanation of reactive programming is that variables are monitored for changes in values and trigger any number of computations; in interactive interfaces, this could mean that when a slider is moved, the new value will trigger a function that will then be used to update the interface again, creating a loop that works very well with interactive visualizations. In Python, *Streamlit* (https://streamlit.io) is a package that enables the creation of interactive web apps in a very similar paradigm. Figures 1 and 2 show how two simple interfaces can be created using Plotly in R and Streamlit in Python with a minimal amount of programming.

Other frameworks embed the ease to design an interface into the environments where people code. For example, the *voila* (https://github.com/voila-dashboards/voila) framework assembles dashboard-like interfaces from Jupyter Notebook documents. In R, a similar package is *flexdashboard* [SIAB22], which assembles an interface from an R Markdown document. Some of the packages described above also contain dashboard and layout functionalities, such as *Bokeh* with *grid* functions, or are used as the drawing package for dashboards, such as how plotly is used in the *Dash* framework (https://dash.plotly.com).

In the context of data visualization software, building CMV-based applications is one aspect of translating research into software; the visualizations themselves are also the key elements of an application. One benefit of the frameworks discussed above is that outputs from different packages can be easily integrated into an application. For designers, this means that they can develop visualizations as independent components and publish those components as stand-alone packages if they wish so. This is also applicable to novel visualization techniques, which can include custom methods for rendering as well.

4.3. Interactivity

For individual visualization components, there are two sides to interaction: direct user manipulation of the visualization and the communication of interactions between components or an interface. For web-based components, user manipulation is normally done through events that are controlled in JavaScript. If using any of the frameworks above, programmers will have access to objects that contain a value representing a state (e.g. a button has been clicked) of the interface and, depending on the level of integration with a visualization package, the parameters of a selection. The Shiny framework, for example, can be integrated with Vega plots through the *vegawidget* [LV22] package. For visualizations that are not web-based, interactions must be dealt directly with the graphical system that is being used and may require additional work.

4.4. Scalability

Another challenge for the creation of visualization packages is scalability, which can come in many forms, such as algorithmic, computational and visual [RPA*22], for example. For computational issues such as time to compute results, researchers have to deal with hurdles related to the language of choice. In both Python and R, it is common for developers to write functions in variants of C/C++ that are faster than if written in the former languages. Visualization researchers have tried to address such issues through more

efficient computation performance [MHH19] to progressive visualization [LM19, ZGC*17], but some of the solutions are in developing stages or sometimes are difficult to implement in a language of choice. The Shiny framework, for example, supports interaction delays, i.e. waiting for all interactions to stop before triggering a computation, but it does not have a method to stop a visualization from being rendered once that step is triggered.

On the computation side, support for solutions such as multithreading and parallel computations vary across the languages. R, for example, does not natively support parallel computations, thus programmers must rely on packages. However, some of the widely used packages work differently across operationg systems, adding extra work for programmers as well.

Issues with data size and rendering time must also be dealt with by programmers implementing visualizations or full systems. Regarding data size, there may also be issues with memory limits for loading datasets or even computation on remote servers. Both ecosystems include support for modern databases and filesystems, as well as packages that support the creation of APIs to facilitate data access. With rendering time, programmers must be aware of the speed and toll of computation on users' machines with web-based outputs, which are typically rendered on the user's browser and not on the Python or R backends. The use of JavaScript packages that speed up rendering can also help with that – in this case, a trade-off to be considered is the number of packages loaded in JavaScript in addition to the overheads of Python and R.

4.5. Novel visualization designs

For web-based visualizations, an appealing aspect of both ecosystems is the ability to reuse visualization components that were developed using libraries such as D3.js and web application frameworks such as React or Vue. In R, it is possible to create R-friendly versions of components downloaded from the npm registry, which is the de-facto standard repository for JavaScript modules. For Python, the Dash framework contains functions and templates to facilitate that. A pipeline that begins with the deployment of a visualization component as a JavaScript module can, in the end, encompass the creation of packages for both Python and R.

Personal perspective: as an exercise, I have successfully *ported* some interactive visualizations from the SHAP Python package for machine learning explainability into an R package (https://github.com/rhenkin/rforceplots), using the *reactR* package. This was successful due to the fact that the authors of SHAP published the D3-based interactive visualizations separately on npm.js (https://www.npmjs.com/package/shapjs). I was therefore able to create an API that enables using those visualizations with native R packages, while removing Python dependency.

5. Addressing software issues and obstacles to successful package creation

This section describes how the creation of packages can help to address issues identified inside and outside the visualization research community that are related to software in research. As the creation of packages is not a silver bullet that will solve all problems, obstacles and potential community and individual actions are also described.

5.1. Software development

5.1.1. Creating reusable software

Researchers have described the need for visualization building blocks to facilitate new research. Existing blocks have been identified [RCM*20] mostly as tools for visualization design, but, in the form of packages, they could include smaller, reusable components that directly tie visualizations to other pieces of software, such as machine learning models. In this case, an implementation challenge is to build visualizations that are not only the endpoint of a workflow but can also be combined with each other. For example, an interactive visualization package for model steering could be designed to receive a model results object as an input, and the visualization interactions could be linked to a model interpretability package. Following this example, a complete visual analytics system can be built using existing and new components based on frameworks such as Shiny and Streamlit. Reusing packages could also create a trust loop: the more a package is *seen* to be used, the more researchers will be encouraged to use it. The development of building blocks is also related to some of the issues discussed further down, such as encouraging new researchers by providing better tools to begin projects.

5.1.2. Testing packages

Visualization research does not typically include the verification of published software, and prototypes that are not developed following software engineering practices can quickly become bloated and difficult to test. In the R ecosystems, packages such as *testhat* [Wic11] and *RUnit* [BJK18] facilitate the creation of tests to check that code works, that visual outputs are as expected and Shiny app interfaces behave as intended. Including tests is not a requirement for either the CRAN repository or the Python repositories, but the tight integration of these tools with package development is an encouragement for developers to include testing from the outset of software creation. In Python, the *unittest* package is included in all installations as part of the Python Standard Library and other packages are also available. Source code repositories such as GitHub also allow the creation of automated scripts that trigger testing when there are changes in the code, ensuring that uploaded packages are passing the defined tests. These scripts also support testing the requirements for repositories such as CRAN.

5.1.3. Maintaining packages

Identifying and fixing errors in software, communicating with users and updating imported packages are crucial for well-maintained software. Any software uploaded to the CRAN repository is connected to other software that it depends on, such that, for each package, there is also a list of *reverse dependencies*. This enables developers to contact and notify other users of that package regarding breaking changes or errors that may have been identified. On websites like GitHub, it is also possible to set up automated alerts for vulnerabilities in dependencies, as well as workflows that test the installation of dependencies, to check if any updates in the dependencies causes the software to stop working.

Authors of successful packages that are used outside the visualization community can also engage with sponsorship programs,

widely used in Python and R and development, to dedicate time to improving or maintaining a package. Typically, package authors identify issues or improvements that are eligible for sponsorship and interested parties can fund the required work.

5.2. Beyond packages

5.2.1. Code and software peer review

Code review is an established practice of software development where code is inspected and critiqued by other members of a team to improve it, either for finding problems or improving documentation and creating alternative solutions to problems [BB13]. In research teams, "code clubs" can be set up and run by researchers and students to introduce reviewing practices. These clubs can also be run across teams and departments as well, and include both visualization and non-visualization researchers. When focused on package development, there are aspects such as successful installation and good documentation that code reviewing can help in addition to automated testing.

Personal perspective: I have been involved in a code review club within a computational biology interest group but run across departments, which is in its third year running. Colleagues have found benefits from interacting with other people's codes and receiving feedback, but code complexity was also an intimidating factor. Another obstacle to a successful code review is having enough volunteers offering their code to review to keep the activities going. In the computational biology group, where producing software is not actually the aim of most participants, this may be more problematic than in a visualization software code club.

Outside research groups, there are also organizations that provide and encourage software peer review, such as rOpenSci (http://ropensci.org) and pyOpenSci (http://https://www.pyopensci.org), which also incentivize the creation of reusable software and best practices for publishing software. Although visualization software is currently not within the scope of rOpenSci, for example, they are still good examples that could eventually make their way into the visualization research community, and can also inspire smaller research teams.

5.2.2. Hackathons and other events

Hackathons are events where small teams work intensively on software projects in a short period of time (typically no more than 3 days). They cover specific applications and technologies or also subjects, e.g. transport systems [BM14], and may also be employed in public spaces and learning environments [NM16]. Although aimed at prototyping, these events could lead to medium and longer-term benefits. For visualization research, hackathons could encourage researchers that work on similar technologies to solve problems together and collaborate. One possibility would be to encourage people to work on visualization challenges such as the VAST Challenge and SciVis context with a hackaton perspective.

Another type of event that could help to improve software development is the annual Hacktoberfest (https://hacktoberfest.com), a month-long initiative to encourage people to contribute to open-source software on GitHub, by improving documentation or attempting to solve logged issues. Similarly to hackathons, this type of event could serve as an inspiration for teams to spend time improving their visualization software.

5.3. Obstacles for package creation

5.3.1. Incentives

As in other fields, there are also several incentives-related obstacles that prevent researchers from improving software, no matter its form. One of them is the recognition of software as a research output, as mentioned above. For people planning to follow an academic career, time spent on software development will not always be rewarded. Venues such as the Journal for Open Source Software (JOSS) provide an opportunity to get more recognition for smaller pieces of software, but researchers must also consider the purpose of the software, which could be used to demonstrate contributions to visualization research. Organizations for RSE have been pursuing more recognition for software and visualization researchers could benefit from getting involved in those as well. The practice of citing software used in research rather than only adding external links to web pages also helps with that.

Personal perspective: I have published an article in the JOSS for an R package that contains Shiny app for hierarchical clustering [HB22]. As the main efffort was about tying up together separate packages rather than a research-oriented development, I thought the JOSS was a good venue to formally publish it. It encouraged me to polish the package as much as possible and consider many different use cases. In my case, it also worked as a way to wrap up the project as I had no further plans for the tool.

5.3.2. Onboarding of new researchers

Onboarding, the integration of new members into an organization, is a fundamental aspect of open-source software (OSS), and can also affect package development and the visualization community. No matter the level of experience in programming in a particular language, programmers will face unknown code and collaborators with different goals, experience and skills, and may end up being discouraged from contributing. As the long-term survival of OSS projects depends on the constant rotation of developers, researchers have suggested guidelines [STG19] to facilitate the onboarding process, also identifying personal, communication and technical obstacles for the success of onboarding. Research teams also have similar needs: one or more group leaders may remain constant over time, whereas post-doctoral researchers, PhD students and research assistants move on to different positions, places and interests. For visualization software to be successful, research teams should also consider onboarding best practices.

5.3.3. Cultural factors in applied domains

An obstacle to reaching wider audiences for visualization software is that cultural factors may hinder adoption. As many visualization researchers know very well, it is very difficult to break with established practices, even when those practices can be improved. Visualization packages must conform to those, which may range from data formats to visualization designs. On the software side, to increase adoption, thorough examples and on-screen guidance can help new users. R package development strongly encourages

the use of *vignettes* to demonstrate functionality, and web apps can include on-screen tutorials to facilitate use. Software contributions are also often more valued if they enable new scientific findings in the domain; non-scientific findings such as more efficient workflows would likely be directed to a short "application note" (e.g. Bioinformatics journal), which may well undervalue the effort put by researchers into the software and disincentivize future work.

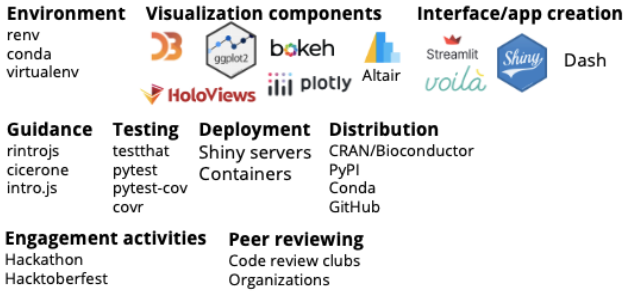

Figure 3: *Summary of packages and activities described throughout the paper that facilitate creating visualization software.*

5.4. Conclusion

The paper described the Python and R ecosystems in the context of information visualization software, useful tools, existing infrastructure and initiatives, as summarized in Fig 3. The path towards better software includes actions readily available for researchers as well as community-level actions that require more coordination and time. Raising researchers' awareness of *what is out there* is also aimed at helping the community move beyond individual efforts.

However, this paper also acknowledges that the situation suggested here does not always align with some overarching aims, despite the ambition for better software. For example, for researchers looking into commercializing their outputs, there might be issues about how to break up their code so that parts of it are made open-source and others are not. Other researchers may be interested in creating better software but not to the point of investing effort and time into creating packages. Another aspect of commercial outputs is the level of polish required for tools, and the different incentives and personnel involved, none of which are discussed in this paper.

This paper also covered visualization software support from the information visualization perspective. For scientific visualization, which may have more demands on computation and rendering, not every solution presented on the software side will be applicable.

Finally, the paper used Python and R as examples of ecosystems where data science activities happen. The Julia language is already a relatively established alternative, in part due to its native support of features such as parallelism. At the same time, the visualization community is still strongly engaged with web-based visualizations based on JavaScript. The software engineering aspects described in this paper should be also applicable to those languages but they could merit other similar articles as well.

Acknowledgements

For this work, RH was funded by the Health Data Research UK (grant ref: LOND1).

References

[ABD*21] ANZT H., ET AL.: An environment for sustainable research software in Germany and beyond: Current state, open challenges, and call for action. *F1000Research* (2021). doi:10.12688/f1000research.23224.2. 2

[Bar10] BARNES N.: Publish your computer code: It is good enough. *Nature* (2010). doi:10.1038/467753a. 2

[BB13] BACCHELLI A., BIRD C.: Expectations, outcomes, and challenges of modern code review. *Proceedings of the 2013 International Conference on Software Engineering* (2013). 7

[BCHK*22] BARKER M., ET AL.: Introducing the FAIR Principles for research software. *Scientific Data* (2022). doi:10.1038/s41597-022-01710-x. 2, 3

[BJK18] BURGER M., ET AL.: *RUnit: R Unit Test Framework*, 2018. 6

[BM14] BRISCOE G., MULLIGAN C.: *Digital Innovation: The Hackathon Phenomenon*. Tech. Rep. 6, 2014. 7

[BOH11] BOSTOCK M., ET AL.: D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* (2011). doi:10.1109/TVCG.2011.185. 5

[Bok18] BOKEH DEVELOPMENT TEAM: *Bokeh: Python Library for Interactive Visualization*, 2018. 5

[CCA*22] CHANG W., ET AL.: *Shiny: Web Application Framework for R*, 2022. 5

[CHH*13] CROUCH S., ET AL.: The Software Sustainability Institute: Changing Research Software Attitudes and Practices. *Computing in Science & Engineering* (2013). doi:10.1109/MCSE.2013.133. 2

[CKB*21] COHEN J., ET AL.: The Four Pillars of Research Software Engineering. *IEEE Software* (2021). doi:10.1109/MS.2020.2973362. 2

[CLG17] CONWAY J. R., ET AL.: UpSetR: An R package for the visualization of intersecting sets and their properties. *Bioinformatics* (2017). doi:10.1093/bioinformatics/btx364. 1

[Gau08] GAUTIER L.: *Rpy2*, 2008. 4

[GCL21] GADHAVE K., ET AL.: *Reusing Interactive Analysis Workflows*. Preprint, Open Science Framework, 2021. doi:10.31219/osf.io/udqjr. 3

[Har18] HAROZ S.: Open Practices in Visualization Research : Opinion Paper. In *IEEE Evaluation and Beyond - Methodological Approaches for Visualization (BELIV)* (2018), IEEE. doi:10.1109/BELIV.2018.8634427. 1, 2

[HB22] HENKIN R., BARNES M. R.: visxhclust: An r shiny package for visual exploration of hierarchical clustering. *Journal of Open Source Software* (2022). doi:10.21105/joss.04074. 7

[HCH*20] HASSELBRING W., ET AL.: From FAIR research data toward FAIR and open research software. *it - Information Technology* (2020). doi:10.1515/itit-2019-0040. 1, 2

[Hun07] HUNTER J. D.: Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* (2007). doi:10.1109/MCSE.2007.55. 4

[IHG12] INCE D. C., ET AL.: The case for open computer programs. *Nature* (2012). doi:10.1038/nature10836. 2

[ISLH19] IBRAHIM S., ET AL.: Interactive in situ visualization and analysis using Ascent and Jupyter. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2019). doi:10.1145/3364228.3364232. 3

[JKA*17] JIMÉNEZ R. C., ET AL.: Four simple recommendations to encourage best practices in research software. *F1000Research* (2017). doi:10.12688/f1000research.11407.1. 1

[LM19] LI J. K., MA K.-L.: P5: Portable Progressive Parallel Processing Pipelines for Interactive Data Analysis and Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2019). doi:10.1109/TVCG.2019.2934537. 6

[LV22] LYTTLE I., VEGA/VEGA-LITE DEVELOPERS: *Vegawidget: 'htmlwidget' for 'Vega' and 'Vega-Lite'*, 2022. 5

[MA22] MURDOCH D., ADLER D.: *Rgl: 3D Visualization Using OpenGL*, 2022. 4

[MHH19] MORITZ D., ET AL.: Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (2019). doi:10.1145/3290605.3300924. 6

[MQB19] MING Y., ET AL.: RuleMatrix: Visualizing and Understanding Classifiers with Rules. *IEEE Transactions on Visualization and Computer Graphics* (2019). doi:10.1109/TVCG.2018.2864812. 1

[MT20] MUNK M., TURK M.: Widgyts: Custom Jupyter Widgets for Interactive Data Exploration with yt. *Journal of Open Source Software* (2020). doi:10.21105/joss.01774. 3

[NM16] NANDI A., MANDERNACH M.: Hackathons as an Informal Learning Platform. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (2016). doi:10.1145/2839509.2844590. 7

[NSS21] NARECHANIA A., ET AL.: NL4DV: A Toolkit for Generating Analytic Specifications for Data Visualization from Natural Language Queries. *IEEE Transactions on Visualization and Computer Graphics* (2021). doi:10.1109/TVCG.2020.3030378. 1

[OCL*21] ONO J. P., ET AL.: *PipelineProfiler:* A Visual Analytics Tool for the Exploration of AutoML Pipelines. *IEEE Transactions on Visualization and Computer Graphics* (2021). doi:10.1109/TVCG.2020.3030361. 1

[RCM*20] REINA G., ET AL.: The moving target of visualization software for an increasingly complex world. *Computers & Graphics* (2020). doi:10.1016/j.cag.2020.01.005. 1, 6

[RPA*22] RICHER G., ET AL.: Scalability in visualization. *IEEE Transactions on Visualization and Computer Graphics* (2022). doi:10.1109/TVCG.2022.3231230. 5

[RSB*22] RUDIGER P., ET AL.: Holoviz/holoviews: Version 1.15.2. Zenodo, 2022. doi:10.5281/ZENODO.7277284. 5

[SIAB22] SIEVERT C., ET AL.: *Flexdashboard: R Markdown Format for Flexible Dashboards*, 2022. 5

[SMB*16] STODDEN V., ET AL.: Enhancing reproducibility for computational methods. *Science* (2016). doi:10.1126/science.aah6168. 2

[SO20] SCHMIDT J., ORTNER T.: Visualization in Notebook-Style Interfaces. *VisGap - The Gap between Visualization Research and Visualization Software* (2020). doi:10.2312/VISGAP.20201104. 3

[STG19] STEINMACHER I., ET AL.: Let Me In: Guidelines for the Successful Onboarding of Newcomers to Open Source Projects. *IEEE Software* (2019). doi:10.1109/MS.2018.110162131. 7

[UAT22] USHEY K., ET AL.: *Reticulate: Interface to 'Python'*, 2022. 4

[VGH*18] VANDERPLAS J., ET AL.: Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software* (2018). doi:10.21105/joss.01057. 1, 5

[WDA*16] WILKINSON M. D., ET AL.: The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data* (2016). doi:10.1038/sdata.2016.18. 1, 2

[WHS20] WU Y., ET AL.: B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (2020). doi:10.1145/3379337.3415851. 3

[Wic11] WICKHAM H.: Testthat: Get started with testing. *The R Journal 3* (2011), 5–10. 6

[Wic16] WICKHAM H.: *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. 4

[ZGC*17] ZGRAGGEN E., ET AL.: How Progressive Visualizations Affect Exploratory Analysis. *IEEE Transactions on Visualization and Computer Graphics* (2017). doi:10.1109/TVCG.2016.2607714. 6