

How do I select an R package for my clinical workflow?

Sean Lopp & Phil Bowsher, RStudio; Boston, USA

Abstract

At the start of most R implementations, questions pertain to packages - how to install, manage, update, create etc. I get a similar question regularly which is the basis of this paper...

How do I select an R package for my clinical workflow?

This question has a few layers that are important to understand.

Usually this question is not asked by R users but often IT admins of the environment. These system administrators do not often know R well and are not familiar with the R ecosystem. Sometimes these people are R Admins (people specifically designated to maintain the R environment) and the conversation is a little easier.

Introduction

This paper is for administrators that are new to R and have been assigned with implementing an R environment used for clinical reporting and workflows. For the focus of this paper, information will be provided about the use of R on a server and not on a desktop computer.

My advice below is just that, advice, and not a framework for selecting R packages. My hope is that my experiences can help guide you along your path of implementing R and that further direction will grow from the "R in pharma" community. I will highlight in the paper further work being done in the R community to help support package validation.

Below I will discuss the package landscape in R and some direction for selecting packages specifically pertaining to clinical workflows. If you are unfamiliar with R, *then you should read the appendix first for important background.*

How do I select an R package for my clinical workflows?

The number of R packages published to CRAN has been steadily increasing. In the first 10 years of R, there were roughly 1,000 packages on CRAN. In the second 10 years, there were nearly 13,000. This scope and volume of packages makes R a wonderful ecosystem for research, statistics, and data science while also introducing a need for people to be able to find and utilize packages in regulated environments. The R code below will list the total number of packages on CRAN:

```
nrow(available.packages())
```

```
> nrow(available.packages())  
[1] 13526
```

Most clinical workflows follow a process similar to the [Data Science workflow](#) - importing, cleaning and transforming data, building visualizations and models and ultimately communicating the results via reports or applications.

The link below outlines many of the packages needed that follow the data science workflow:

<https://github.com/rstudio/RStartHere>

However, most clinical workflows will not have all of these packages, and most will require packages that are not on the list. How do you go about picking packages that will be reliable, performant, and correct - all over time?

Below are some questions that can help inform a discussion on which packages to use.

Does the package have an active author or development team?

The CRAN page for the package will list the Authors. Below is an the CRAN page from ggplot2, a popular R package that provides a system for declaratively creating graphics, based on The Grammar of Graphics.:

<https://cran.r-project.org/web/packages/ggplot2/index.html>

If the package source is stored on GitHub, you can find the active developers and maintainers by searching the GitHub Insights tab:

<https://github.com/tidyverse/ggplot2/graphs/contributors>

The top 10 github contributors for ggplot2 are:

- #1 hadley 2,258 commits
- #2 wch 589 commits
- #3 kohske 194 commits
- #4 thomasp85 76 commits
- #5 jiho 62 commits
- #6 lionel- 54 commits
- #7 karawoo 45 commits
- #8 clauswilke 44 commits
- #9 BrianDiggs 25 commits
- #10 yutannihilation 23 commits

Is the package well maintained?

In addition to looking at the number of authors, you will want to note how active the authors are in answering questions and improving the package. One of the top places to look first is the number of releases of a package. Below is the CRAN and github pages for ggplot2 showing releases:

<https://cran.r-project.org/src/contrib/Archive/ggplot2/>

<https://github.com/tidyverse/ggplot2/releases>

Moreover, the NEWS page can provide a lot of good information as seen below for ggplot2:

<https://cran.r-project.org/web/packages/ggplot2/news/news.html>

It is also advised to review the issues and pulls on github - check to see if authors are responsive to answering questions or addressing bug reports:

<https://github.com/tidyverse/ggplot2/issues>

<https://github.com/tidyverse/ggplot2/pulls>

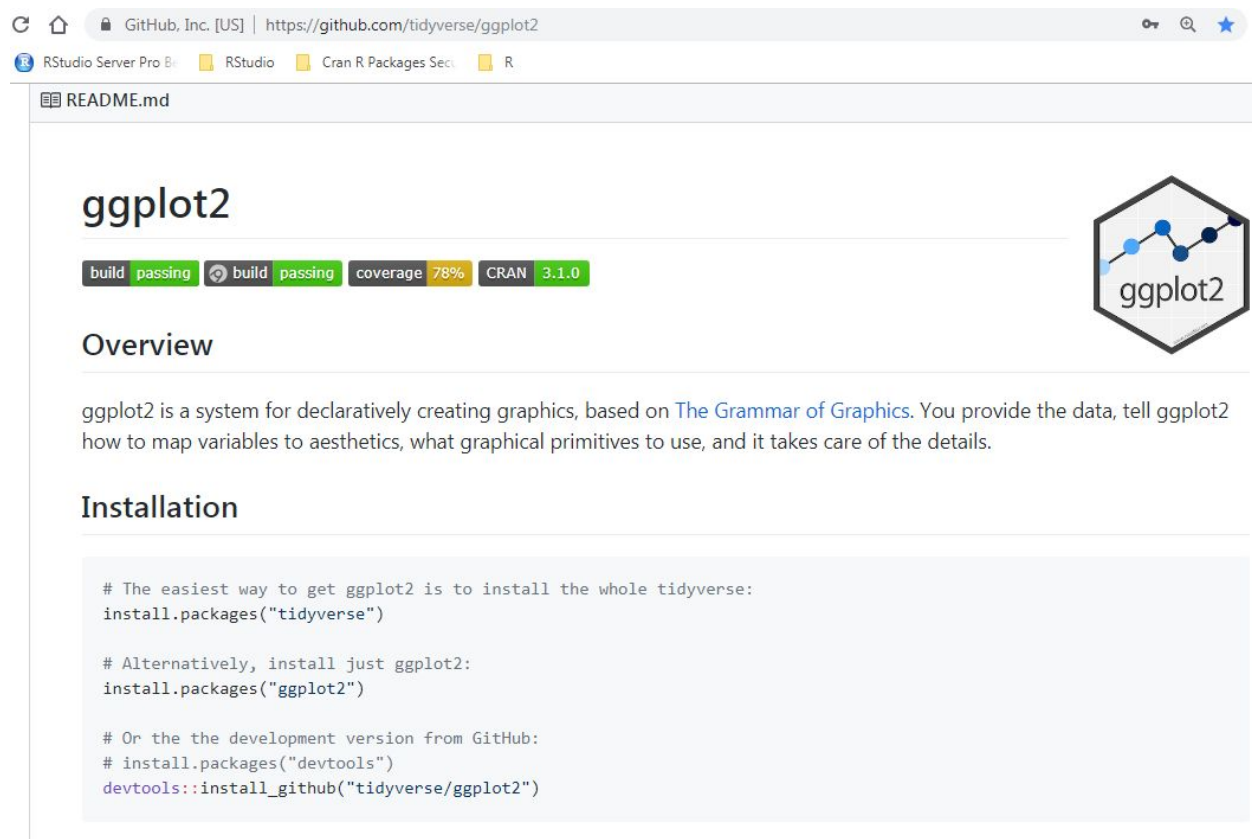
Unfortunately, just looking at the number of releases or the date of the last release does not paint the whole picture. Some packages will have lots of recent releases because they are rapidly changing - it is unlikely those are a good fit for a validated environment. Other packages might not have had a release for quite some time - is this because the package has been abandoned? Or is it because the package is really stable? Evaluating the package's state of life as it relates to the package development life cycle is a helpful way to answer these questions.

Please see the link below for information on package life cycle.

<https://www.tidyverse.org/lifecycle/>

Is the package on CRAN?

Many great packages will go through a series of testing by power users in the community. Typically during this stage, packages are available only on GitHub. When ready, the package author will submit the package to CRAN, which is the primary source of R packages around the world. Once a package is available on CRAN, the package is available through the **install.packages** command and the github documentation will normally look like this:



ggplot2

build passing build passing coverage 78% CRAN 3.1.0

Overview

ggplot2 is a system for declaratively creating graphics, based on [The Grammar of Graphics](#). You provide the data, tell ggplot2 how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

Installation

```
# The easiest way to get ggplot2 is to install the whole tidyverse:
install.packages("tidyverse")

# Alternatively, install just ggplot2:
install.packages("ggplot2")

# Or the the development version from GitHub:
# install.packages("devtools")
devtools::install_github("tidyverse/ggplot2")
```

Above is the github page for ggplot2 and you can see it is on CRAN with ggplot2 version 3.1.0. You can see a version of a package by running the following code in R:

```
> packageVersion("ggplot2")
[1] '3.1.0.9000'
```

Before CRAN accepts a package, CRAN runs a thorough set of tests to ensure the package will work with other packages on CRAN. Getting a package through these checks ensures the package is stable, and also indicates the package author is serious and motivated. More information on tests can be reviewed at the link below:

<http://r-pkgs.had.co.nz/tests.html#test-cran>

How many times has the package been downloaded?

You can use the `cranlogs` package to download package information. `ggplot2` has been downloaded over 20 million times.

<https://cran.r-project.org/web/packages/cranlogs/index.html>

The `tools` package can also be used:

<https://stat.ethz.ch/R-manual/R-devel/library/tools/html/CRANtools.html>

`metacran/crandb` can also be used: <https://github.com/metacran/crandb#the-raw-api>

Below is an example workflow of analyzing package information such as downloads data:

<https://juliasilge.com/blog/scraping-cran/>

Does the package have documentation?

Documentation is important as it highlights that a package is well maintained and supported. Documentation for a package can come in a few forms.

Website: <https://ggplot2.tidyverse.org/>

Updates: <https://www.tidyverse.org/articles/2018/05/ggplot2-2-3-0/>

Cheat Sheet: <https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf>

README: <https://cran.r-project.org/web/packages/ggplot2/readme/README.html>

Vignettes: <https://cran.r-project.org/web/packages/ggplot2/vignettes/extending-ggplot2.html>

<https://cran.r-project.org/web/packages/ggplot2/vignettes/ggplot2-specs.html>

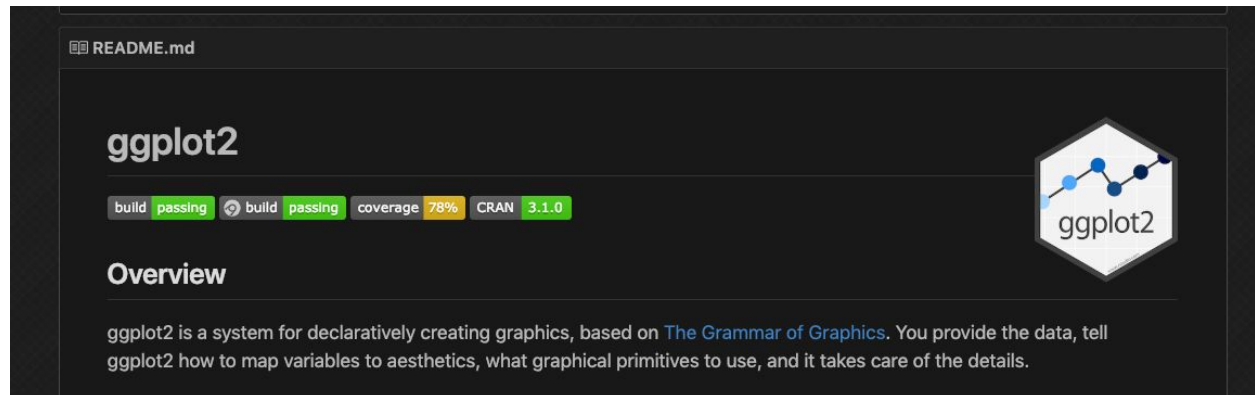
Does the package have tests?

In addition to documentation, a critical indicator that a package is ready for primetime is checking to see whether the package has tests. Normally, package authors include tests in a directory alongside their package code. Here are the tests for `ggplot2`:

<https://github.com/tidyverse/ggplot2/tree/master/tests>

Badges:

Badges report information, such as the CRAN version or test coverage and link out to relevant external resources. Great packages will record the test coverage - a metric that indicates how many of the package's functions have been tested.



ggplot2 has 78% coverage, which means it would be very difficult for the package author to accidentally break important functions!

In addition to preventing breakage, many authors use tests to check that their package performs tasks correctly. Sometimes, the tests written by a package author can be supplemented by others.

For managing test and builds, package authors will use various tools and strategies. Below is a short list of terminology used in this space:

Use this package - package to help with the process of package development

Continuous Integration (CI) - practice of merging all developer working copies to a shared mainline several times a day

Travis - a CI tool used to build and check R packages on Linux

Unit Tests - testing individual units/components of a software are tested

testthat package - package for formal automated testing of packages

Code Coverage - measuring the percentage of code run when testing occurs

covr package - interface for code coverage in R packages

You can compare the test and other information in packages with the `packagemetrics` package.

<https://github.com/ropenscilabs/packagemetrics>

The similar package, `packageMetrics2`, has similar information for compare differences between packages:

<https://github.com/MangoTheCat/packageMetrics2>

Currently, the “R in Pharma” community is working on writing correctness tests for commonly used packages. These supplemental tests will be used to support ongoing validation efforts. Learn more about the PSI AIMS SIG working group at the link below:

Application and Implementation of Methodologies in Statistics (AIMS)

<https://pharmar.org/>

Workshops and talks about the efforts are shared at the R in Pharma conference, yearly at Harvard University in August:

<http://rinpharma.com/>

Below is the PSI AIMS SIG R in Pharma conference presentation from August 2018:

https://www.pharmar.org/presentations/R_Validation_Workshop.pdf

Is there a champion that promotes the package?

Most packages don't have many authors. A critical factor to getting a package known and used by the ecosystem is having a champion that is advocating for the package. This could include maintaining the documentation, giving talks at conferences, or doing online webinars. This may or may not be the package author. Here is an example:

<https://ropensci.github.io/drake/>

<https://wlandau.github.io/drake-talk/#/>

What is the scope of the package?

Another consideration when picking a package is identifying what you want to use the package for. Some packages are small and have very specific purposes. Other packages are much broader and provide many functions. Typically, pick packages where you are confident users will use a majority of the package's functions. To help identify whether this is true, consider analyzing your project with the itdepends package:

<https://resources.rstudio.com/rstudio-conf-2019/it-depends-a-dialog-about-dependencies>

<https://speakerdeck.com/jimhester/it-depends>

What does the package depend on?

Many R packages depend on other R packages. When picking a package, this information can be used in two ways. First, if a package has a large number of reverse dependencies (packages that rely on it), there is a good chance the package is stable. For example, the Rcpp package is used by thousands of other packages:

ABOUT

<http://www.rcpp.org>, <http://dirk.eddelbuettel.com/code/rcpp.html>, <https://github.com/RcppCore/Rcpp>

▶ 2	IMPORTS
▶ 1	DEPENDS
▶ 7	SUGGESTS
▶ 1337	REVERSE IMPORTS
▶ 169	REVERSE DEPENDS
▶ 25	REVERSE SUGGESTS
▶ 1540	REVERSE LINKING TO

Package dependencies can take different forms. Imports, Depends, and Suggests all indicate packages that the current package relies on. The “Reverse” fields are packages that, in turn, depend on the current package.

If you want to use a package that has a lot of dependencies, then you’ll need to think about the stability of those dependencies as well. You can see information on dependencies at the bottom of the packages page on cran. Here is an example for ggplot2:

<https://cran.r-project.org/web/packages/ggplot2/index.html>

Finally, some R packages rely on software that is external to R. These external dependencies are listed in the package’s DESCRIPTION file:

```
Maintainer:      Roger Bivand <Roger.Bivand at nhh.no>
License:         GPL-2 | GPL-3 [expanded from: GPL (≥ 2)]
URL:             http://www.gdal.org, https://r-forge.r-project.org/projects/rgdal/
NeedsCompilation: yes
SystemRequirements: for building from source: GDAL ≥ 1.11.4, library from https://trac.osgeo.org/gdal/wiki/DownloadSource and
PROJ.4 (proj ≥ 4.8.0) from https://download.osgeo.org/proj/; GDAL OSX frameworks built by William
Kyngesburye at http://www.kyngchaos.com/ may be used for source installs on OSX. For installation with older
external dependencies, override configure checks with --configure-args="enable-deprecated=yes". Consider
source installations using archived versions of rgdal contemporary with installed external dependencies, for
example rgdal_0.8-7 for PROJ4 4.8.0 (March 2012).

Materials:       ChangeLog
In views:        Spatial
CRAN checks:     rgdal results
```

<https://cran.r-project.org/web/packages/rgdal/index.html>

If a package has a system requirement, you will need a plan for managing the external software.

Do you have license restrictions?

R packages each have their own software licenses. The license is listed in the package's DESCRIPTION file. For ggplot2, you can see that here:

<https://cran.r-project.org/web/packages/ggplot2/index.html>

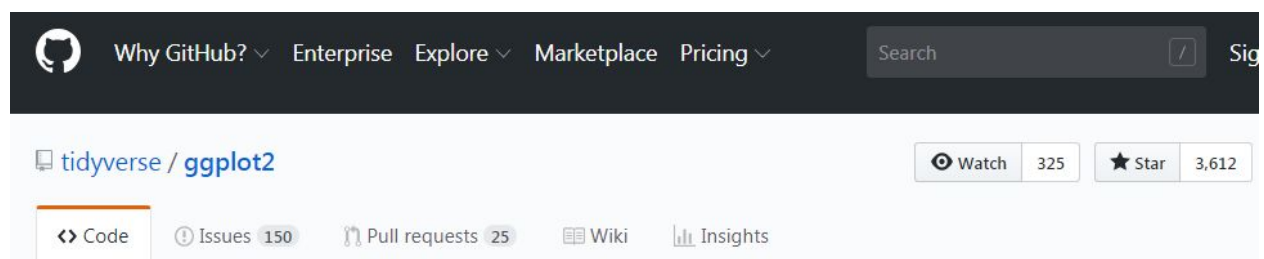
The license is GPL-2. You can see this information in R by running:

```
packageDescription("ggplot2")
```

Is the Package Starred on GitHub?

An important metric is stars on github. This is an overall rating by the community regarding feedback of packages.

ggplot2 has been starred 3,612 times.



Below is an analysis of this data:

<https://stevenmortimer.com/most-starred-r-packages-on-github/>

Conclusion

When you are considering using a package, especially for validated work, it is important to have a conversation about the package's stability, performance, and complexity. The questions outlined in this paper can help inform your package selection, and they can also be used to re-evaluate a project's dependencies overtime.

CONTACT INFORMATION

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

Phil Bowsher

phil@rstudio.com

Sean Lopp

sean@rstudio.com

Supplemental Background & Appendix

Below is a review of some important information that I discover most people do not know.

This section will be organized as follows: we'll talk about what R is, and what make R packages so special. Then I'll discuss where R packages come from, and where they are used in an organization.

CRAN (The Comprehensive R Archive Network) is the world's largest repository of statistical computing language. It is where R lives.

What is R?

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R. To learn more: <https://www.r-project.org/about.html>

Who is behind R?

The source for R and the R base packages are maintained by a closed group of developers called the R Core Team, those who can modify the R source code archive. The recommended packages are maintained by the same group and other select contributors. The current list of members can be found here:

<https://www.r-project.org/contributors.html>

An overview of the history of CRAN can be seen here:

<https://channel9.msdn.com/Events/useR-international-R-User-conferences/useR-International-R-User-2017-Conference/KEYNOTE-20-years-of-CRAN>

What are the R Foundation and the R Consortium?

The R Foundation is a not for profit organization founded by the members of the R Development Core Team. The R Foundation, among other things, works to provide support for the R project and other innovations in statistical computing and hold and administer the copyright of R software and documentation.

The R Consortium, Inc. is a group organized under an open source governance and foundation model to support the worldwide community of users, maintainers and developers of R software. Its members include institutions and companies dedicated to the use, development and growth of R.

How is R distributed, and what is CRAN?

The CRAN site describes CRAN as:

“CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R.”

<https://www.r-project.org/news.html>

The ftp and web servers around the world are often referred to as mirrors and R as “Base R”.

The source for R includes 3 things: R source code, a set of base packages, and a set of recommended packages. All 3 components follow the release and testing cycle outlined here:

<https://www.r-project.org/doc/R-FDA.pdf>

Specifically, the /tests directory of the R source includes tests for R core, the base packages, and the recommended packages: <https://svn.r-project.org/R/branches/R-3-4-branch/tests/>

You can navigate up the URL directory to see the /test folder for different branches of R, eg

<https://svn.r-project.org/R/branches>

CRAN has task views on various information. Some applicable to our focus are here:

<https://cran.r-project.org/web/views/ClinicalTrials.html>

<https://cran.r-project.org/web/views/ReproducibleResearch.html>

What about all the packages I hear about like shiny and ggplot2? Does CRAN create them?

In addition to distributing R, CRAN is the primary package repository in the R community. A team of individuals approves packages **submitted and added to CRAN**. Getting a package added to CRAN is no small feat (pass R CMD CHECK and survive the review process). CRAN maintainers test packages across a matrix of R versions and operating systems. The review process is outlined here:

<https://cran.r-project.org/web/packages/policies.html>

https://cran.r-project.org/web/packages/submission_checklist.html

Any package accepted on CRAN must pass a series of automated tests:

<https://cran.r-project.org/doc/manuals/r-release/R-exts.html#Checking-packages>

The automated tests enforce the CRAN submission policies. The results of these tests are available on CRAN for each package:

http://cran.us.r-project.org/web/checks/check_summary_by_package.html#summary_by_package

https://cran.r-project.org/web/checks/check_results_ggplot2.html

The same tests can be run locally against any package source tar file using a command like:

R CMD check package_0.1.tar.gz

The actual test code is part of R itself, not part of the package being tested. CRAN checks are run for the package plus the latest release of R, patch releases, and the current development branch of R. The tests are run on a variety of operating systems including Linux, Windows, and Mac OS.

In addition to the CRAN checks, many package authors have added their own tests which help ensure that the package's functions work as intended. Most often, these tests are built into the package in a specific way so that whenever the automatic CRAN checks are run the custom tests also run. These additional tests are part of the package source, for more information: <http://r-pkgs.had.co.nz/tests.html#test-cran>

These packages added to supplement base R are often referred to as “contributed packages” and are available for download directly from CRAN.

How do users contribute the open source ecosystem of R?

In addition to creating packages, the video below will cover the strategies and mechanics of making contributions to “mature” open-source projects, such as ggplot2 which is highlighted in this paper and in the video below.

<https://www.rstudio.com/resources/videos/contributing-to-tidyverse-packages/>

What is the RStudio CRAN mirror?

In 2013, RStudio started to maintain its own CRAN mirror, <http://cran.rstudio.com>. The RStudio mirror uses Amazon Cloudfront to maintain copies of CRAN on servers all over the globe. These copies are updated off of the main CRAN mirror in Austria once per day. RStudio created this mirror to provide a consistently fast option around the world, a reliable option for users, and to provide a rich source of data about R and package usage.

RStudio software uses the RStudio CRAN mirror (<https://cran.rstudio.com>) by default.

<https://docs.rstudio.com/ide/server-pro/r-sessions.html#cran-repositories>

The RStudio CRAN mirror is the number one repository used for downloading R and CRAN packages. As of Feb 1 2019, there are over 13,000 packages on CRAN. Information on packages can be downloaded for analysis with this code:

https://github.com/joseph-rickert/RStudio-conf_2018/blob/master/Tidy2.R

How does GitHub come into play here?

While most R packages are shared through CRAN, many R packages are maintained and developed on GitHub prior. This allows users to access them easily through CRAN, but also gives them a chance to view the development activities and transparent access to the source code.

For example, plotly on CRAN is 4.8.0 as of 2/11/2019 as seen by the badge on the github page.

```
> packageVersion("plotly")  
[1] '4.8.0'
```

The github version is 4.8.0.9000. You can see the github package version by viewing the DESCRIPTION file here:

<https://github.com/ropensci/plotly/blob/master/DESCRIPTION>

The latest version of plotly on GitHub can be downloaded with this line of R code:

```
devtools::install_github("ropensci/plotly")
```

This will install the latest development version of plotly and will point to the page:

<https://github.com/ropensci/plotly>

Often users will want to install the github version of a package to access the latest and greatest features or a bugfix.

After installing plotly from github, you can see now it is a later version:

```
> packageVersion("plotly")  
[1] '4.8.0.9000'
```

I find that github installations of a package takes a little more time than from CRAN.

How are packages used or installed?

To install a package from CRAN, the base R function **install.packages** is executed in R. This function installs a package from a CRAN mirror. In R, you can pick which mirror to use. By default, RStudio's CRAN mirror is the first one in the list of mirrors ("0-Cloud"), or if you don't want to select it every time you install a package, you can set it as the default in your .Rprofile:

```
options(repos = c(CRAN = "http://cran.rstudio.com"))
```

After you specify the mirror to use, packages will be installed from that location. For example, if you run `install.packages("ggplot2")` after specifying the RStudio mirror, ggplot2 will be downloaded from that location to your environment.

install.packages has many arguments that can be supplied for specifying what happens when the function is run.

So what actually happens when I run `install.packages("ggplot2")`?

First, a connection is made over HTTP or HTTPS to the specified CRAN mirror. Then a file is downloaded and installed to your environment. The file type is dependent on your operating system. The file is unzipped/unpacked and installed.

Where are R packages installed?

R packages are installed into libraries, which are directories in the file system containing a subdirectory for each package installed there. **These libraries can be common/system libraries or user libraries.** You can see all the library paths installed for your user by issuing the `.libPaths()` command in the R terminal:

```
.libPaths()
```

If missing, the `lib` argument in `install.packages` defaults to the first element of `.libPaths()` for determining the path for installing a package. All paths are used to look for a package when you load them with `library()` with first path priority.

To use the package and its functions, **`library(ggplot2)`** needs to be run in the active R session.

Usually, two steps are required to use an R package:

The package is installed from a repository into a library using `install.packages()`.

The package is loaded from the library for an analysis using `library()`.

Another interesting tool is to run **`tree`** in a shell via RStudio.

<https://support.rstudio.com/hc/en-us/articles/115010737148-Using-the-RStudio-Terminal>

This will list contents of directories in a tree-like format. It is a recursive directory listing program that produces a depth indented listing of files.

What does this error mean *could not find function "%>%"* ?

This is the classic error shown when trying to use a package that has not yet been loaded into your session with `library()` before the code is run.

What is the difference between a package and a library?

Below is a quote from Hadley Wickham, Chief data scientist at RStudio, and instructor of the “Writing functions in R” DataCamp course:

“A package is like a book, a library is like a library; you use `library()` to check a package out of the library” Hadley Wickham (@hadleywickham) December 8, 2014

What is the difference between the system and user library?

This question is often asked by people managing a central linux server used by many people running R code.

A library is tied to a specific version of R; and different versions of R will have different system libraries and require different user libraries and package installations.

What is my working directory?

R is always pointed at a directory on your computer. Often this will be your home directory. When you work within a RStudio project, the working directory will be the head of that directory.

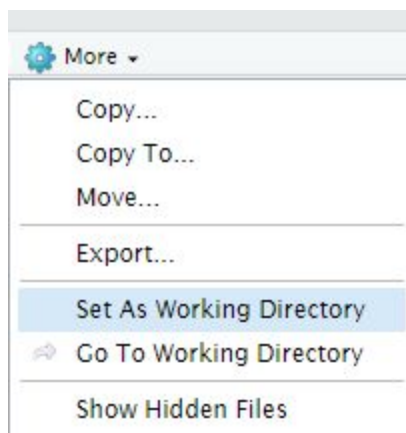
You can find out which directory by running:

```
getwd
```

To change your working directory, use `setwd` and specify the path to the desired folder.

```
setwd(dir)
```

You can also set the working directory within RStudio as shown below:



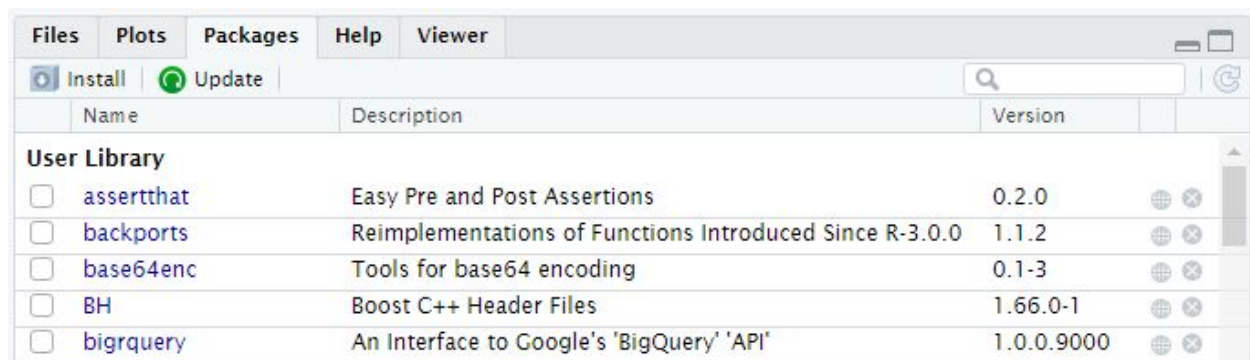
What packages are installed?

You can run:

```
installed.packages()
```

And it will list the Package, LibPath, Version and Priority of the packages you have installed by scanning the DESCRIPTION files of each package found along lib.loc and returns a matrix of package names, library paths and version numbers.

Alternatively, you can view the packages in RStudio via the packages tab:



Both of the methods above will list the packages and version number. In addition, you can run the following command in the R console:

```
packageVersion("packagename")
```

Keep in mind that a library is tied to a specific version of R; and different versions of R will have different package installations. This means if you have multiple versions of R, each of the R installations would have specific package installations. So one package could be installed in 3.3.3 but missing from the 3.4.4 install and running `installed.packages()` would likely be different for each version of R installed.

How do I find information about my R session?

`sessionInfo()` and `devtools::session_info()` are very helpful functions for printing information about your R session. It will list the version R version, Platform, Operating system, Matrix products, locale attached base packages, loaded via a namespace (and not attached).

This information can be written to a file with the lines below:

```
writeLines(capture.output(sessionInfo()), "sessionInfo.txt")
```

```
writeLines(capture.output(devtools::session_info()), "devsessionInfo.txt")
```

How do I install older versions of packages:

You may need to install an older version of a package if the package has changed in a way that is incompatible with the version of R you have installed, or with your R code. Moreover, you may need an older package for deployed content to a separate server.

This can be done with `devtools` and an example is below:

```
require(devtools)
install_version("ggplot2", version = "0.9.1", repos = "http://cran.us.r-project.org")
```

If you know the URL to the package version you need to install, you can install it from source via `install.packages()` directed to that URL. If you don't know the URL, you can look for it in the CRAN Package Archive - <https://cran.r-project.org/src/contrib/Archive/> .

```
packageurl <- "http://cran.r-project.org/src/contrib/Archive/ggplot2/ggplot2_0.9.1.tar.gz"
install.packages(packageurl, repos=NULL, type="source")
```

Or via the commands below:

```
wget http://cran.r-project.org/src/contrib/Archive/ggplot2/ggplot2_0.9.1.tar.gz
R CMD INSTALL ggplot2_0.9.1.tar.gz
```

<https://support.rstudio.com/hc/en-us/articles/219949047-Installing-older-versions-of-packages>

What is Bioconductor?

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. R and the R package system are used to design and distribute Bioconductor software.

Use the `biocLite.R` script to install Bioconductor packages. Just like CRAN, Bioconductor has a repository of packages for download.

Background Continued: Strategies

Now that we know about packages, where they come from, and where they are used, let's look at some strategies organizations can adopt to control each of these components.

How do I handle installing and upgrading R?

Over time, your organization will want to use multiple versions of R. Instead of installing R from a `apt-get` or `yum` repository, we recommend building R from source and maintaining multiple versions of R side by side. This strategy preserves past versions of R so you can manage upgrades and keep your code, apps, and reports stable over time.

Instead of upgrading your existing version of R, a better solution to these problems is to run multiple versions of R side by side. This strategy preserves past versions of R so you can manage upgrades and keep your code, apps, and reports stable over time.

Building R from source is an administrative effort done by someone familiar with compiling code from source. Here is a resource that outlines the necessary steps:

<https://support.rstudio.com/hc/en-us/articles/360002242413-Multiple-versions-of-R>

How can I manage a repository?

Some organizations do not allow CRAN downloads and prefer to have a local repository internal to the organization. Below are some of the current practices and strategies I see utilized by organizations. For validated environments, permissions to install packages to the user library is usually disabled or used in conjunction with a internal repository.

What if my organization is offline?

First, see if your networking team will allow outbound access to CRAN, by enabling access to <https://cran.rstudio.com>. If you are not able to access an online repository, you will need to create and administer your own package repository.

What is an internal repository?

A repository is a directory containing uninstalled R source files or platform-specific binaries. A repository contains a `PACKAGES` file with important information about the repository's content.

There are many reasons why organizations would prefer to use their own repository of packages:

1. An organization operates within an intranet that does not have access to the internet.
2. A validated environment that has strict controls on package versions used.
3. The organization wants to share internal packages across people.

When these situations arise, organizations will use a server internal to the organizations to share and distribute R packages, often the same packages shared by CRAN. Some organizations will mirror the cran repo as explained here:

<https://cran.rstudio.com/mirror-howto.html>

In the mirror link, note that you don't need to follow all of it (since a bunch of that is just how to get added as an official mirror). It is just the easiest way to grab all the packages and files from CRAN at once.

<https://cran.r-project.org/doc/manuals/R-admin.html#Setting-up-a-package-repository>

Once the repository is created, you can set your repo to point to that new location as described here:
<http://docs.rstudio.com/ide/server-pro/r-sessions.html#cran-repositories>

Commercial tools are also available for creating repositories like the RStudio Package Manager.

What is miniCRAN?

In the instructions above, an organization creates an entire copy of CRAN. What if you only want to allow part of CRAN? The miniCRAN package helps you create an internal repository that only contains part of CRAN. It makes it easier to specify a list of CRAN packages you want in that repository. There are other options for creating subsets of CRAN as well, such as RStudio Package Manager.

<https://github.com/andrie/miniCRAN>

What if I have multiple internal repos?

If you have multiple internal repositories, you can tell **install.packages** to look for packages in all of them with the following code:

```
# For 1 Repo
options(repos = c(CRAN = 'http://repo.example.com'))
```

```
# For Multiple Repos
options(repos = c(CRAN = 'http://repo.example.com', REPO2 = 'http://repo2.example.com'))
```

To persist this option, users or admins can place this code snippet inside their .Rprofile or the Rprofile.site file. RStudio 1.2 will directly support configuring this option through a menu instead of having to run R code or place R code in the Rprofile.

What are internal packages?

If you work in an organization that builds R packages with intellectual property, you may not want to share your packages on CRAN or on a public Github repos. Instead, you may want to package your code as an internal package.

How do I build an internal package?

Information on creating a package can be seen here:

<https://www.rstudio.com/resources/videos/you-can-make-a-package-in-20-minutes/>

<https://support.rstudio.com/hc/en-us/articles/200486508-Building-Testing-and-Distributing-Packages>

How can I test an internal package?

Earlier we described the testing options available for CRAN packages. You can apply the same tests to your own internal packages. Often this is accomplished by storing the internal package in a Git repository, and then having a continuous integration tool like Jenkins run R CMD check following code commits. For more information, <https://community.rstudio.com/t/how-do-you-test-your-internal-packages-travis-jenkins-by-hand/13002/7>

How do I distribute an internal package?

Once you have created an internal package and tested it, you can share it using the same internal repository you might have created to help users access CRAN packages.

Managing libraries

Once your organization has a plan for package access (either through CRAN or an internal repository), you will need a strategy for managing installed packages. Organizing packages can take a fair amount of administrative effort, especially when multiple versions of R exist across multiple servers. Below are some resources on this topic:

<https://support.rstudio.com/hc/en-us/articles/215733837-Managing-libraries-for-RStudio-Server>

<https://cran.r-project.org/doc/manuals/r-release/R-admin.html#Managing-libraries>

First, it is important to understand that there are normally three types of libraries: system libraries, user libraries, and project libraries.

System libraries are managed by administrators and contain the standard and recommended packages. Any package installed in the System location will be available to all users on that server. Many organizations encourage users to only use the system libraries. Administrators will install packages directly from an online CRAN repository into the system library. In offline environments, the system library can be copied from the online environment. R users access packages from the system library using the `library()` function. The administrator tells R to look in the correct directory by defining `R_LIBS_SITE` in the `Renviron.site` file or using the R function `.libPaths()` in the `Rprofile.site` file.

User libraries are managed by individual users. There are many advantages to having a user library. Users might want a different version of a package than is installed in the common library. It is typically more convenient for users to install new libraries into their home directory than it is to put in a request to have one installed in a shared library. Users often have a lot of packages that are specific to their needs and not shared by other users. It is also a natural place to build, test, and install custom packages. User specific libraries are typically created in the home directory mounted from a shared file system.

Users can install the latest and greatest version of any package into their user library, while falling back onto older package versions installed in the system library if necessary. For example, `ggplot2` 2.0 can be the system default, but Richard could install `ggplot2` 2.3 into his user library. When Richard runs `library(ggplot2)` it will grab 2.3 from his user library by default. If he wants, he can specify `library(shiny, lib = /path/to/system/library)` in which case it will grab 2.0.

Finally, project libraries allow users to create isolated environments for specific projects. Project libraries are the most flexible option, but also the hardest to manage. Tools like `packrat` help users manage multiple project libraries.

Packrat - What it is?

Packrat is an R package that implements a dependency management system for R.

Packrat creates a special kind of directory - a private package library for a given R project or directory.

What is checkpoint?

Checkpoint allows one to install packages from a Microsoft copy of CRAN. Microsoft creates a copy of CRAN each day, and users can pick which day to use. Other tools provide checkpoints as well, such as RStudio Package Manager.

What is a RStudio Project?

RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents. Many people as part of their workflow start new projects or revive old ones.

<https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects>

What about packages in containers?

Containers are one of the most popular solutions for reproducibility. Docker is a great tool for specifying the steps to create a programming environment. Most organizations use Docker alongside of a strategy for package management. For example, you could create an internal repo of validated packages, and install from that repository into the container. A docker container on its own does not give you reproducibility though, because Docker simply executes commands - those commands themselves have to be reproducible.

To give containers a shot, you can install docker and then take a look at the rocker project (R on docker). It is advisable to pair up packrat with Docker for dependency management. Inside a Dockerfile for an R project you will normally see a line that installs R packages, e.g.,

```
RUN Rscript -e 'install.packages(...)'
```

You can read more here:

<https://rviews.rstudio.com/2018/01/18/package-management-for-reproducible-r-code/>

More information on managing packages in containers can be found at the link below by March 2019:

<https://environments.rstudio.com/>