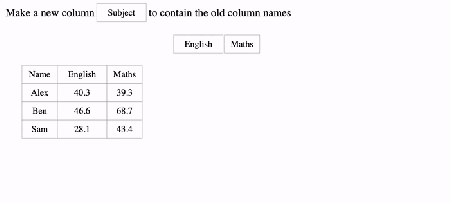
As usual there were many talks that I didn’t get to go to as there are  
around 3~5 tracks across different rooms featuring talks on a certain  
aspect of R such as Shiny, Modelling, Data handling, DevOps, Education,  
etc. In the coming weeks I’ll also add video links to  
the presentations below when they become available from R Consortium’s  
Youtube channel.

Let’s begin!

**Programming**

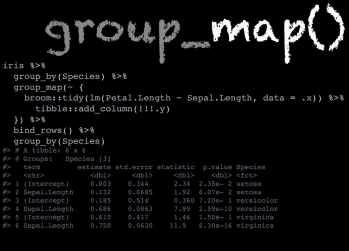
**Enhancements to data tidying: Hadley Wickham**

Acknowledging the difficulty of spread() and gather() you might have heard of the creation of the pivot\_wider() and pivot\_longer() functions in recent  
months.





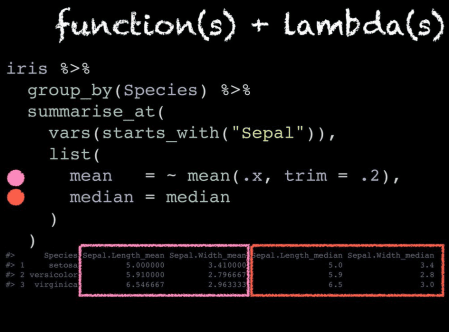
While in previous versions of {dplyr} working in a tidy manner with  
groups was done with group\_by() then dplyr::do(), the latter  
function has been deprecated and have been largely replaced by the  
{purrr} family of functions instead. In this context the group\_map(),  
group\_modify(), and group\_walk() functions iterate like the {purrr}  
functions but instead over groups. You can apply the functions you want  
to apply to each group inline via a lambda, ~ (as below), or you can  
specify a function directly without the lambda.



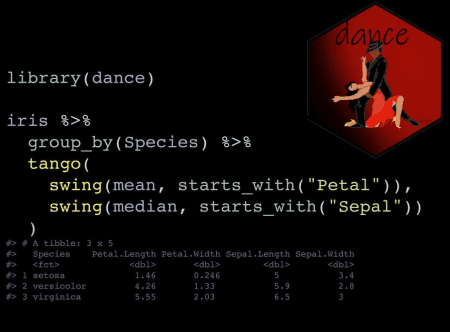
The group\_split() operates similarly to base::split() but splits by  
groups, the output being a list of sliced groups. The group\_keys()  
function returns you the exact grouping structure of the data you used  
group\_by() on, allowing you to check that the structure is right  
before you start applying functions on your data. group\_data() and  
group\_rows() gives you different kind of information about your  
grouped data as can be seen below.


To shorten the group\_by() %>% summarize() workflow you could instead  
use the summarize\_at() function. You can select specific columns with  
vars(), then actions via a lambda, ~, and you can specify multiple  
functions with list().



Romain also talked about the {dance} package which is mainly used to experiment and test out possible new {dplyr} functions by leveraging the  
relatively new {vctrs} and {rlang} packages’ features. The package has a theme of using famous dance moves as the function names!

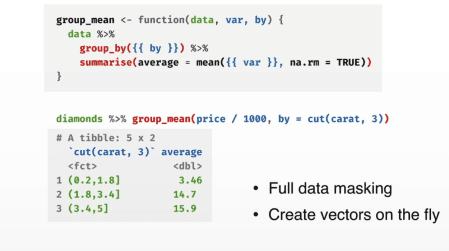


**Reusing tidyverse code**

Let’s talk about programming using {tidyverse} functions. As an  
introduction he went over data masking in {dplyr} and how it is  
optimized for interactive coding and single-use %>%s. The usage of  
non-standard evaluation (NSE) makes analyses easy as you can focus on the data rather than the  
data structure. However, we hit a stumbling block when it comes to when  
we want to create custom functions to program with {dplyr}. This is the  
difference between computing in the work space (as needed) versus  
computing in a data mask.


This is where tidyeval comes into play via {rlang} for flexible and robust programming in the tidyverse. However {rlang} confused a lot of  
people due to the strange new syntax it introduced such as the !!,  
!!!, and enquo(). Also, it introduced new concepts such as  
quasi-quotation and quosures that made it hard to learn for people  
especially with those without a programming background. Acknowledging  
this obstacle, was introduced to make creating tidyeval  
functions easier. The new (read as “curly-curly”) operator was  
inspired by the {glue} package and is a short cut for !!enquo(var).



**Shiny**

**Keynote #2: Shiny apps and Reproducibility – Joe Cheng**

Compared to a R script or R Markdown document, reproducibility suffers  
in Shiny apps as the outputs are transient and **not** archivable.  
RStudio’s talked about how reproducible analysis with Shiny is inconvenient as reenacting the  
user’s interaction steps is necessary. A case for having a simple  
**CLICK** button to view/download a reproducible artifact can be seen in  
various industries such as:

* ex. Drug research/pharma validation (workflow)
* ex. Teaching: statistical concepts and code snippets
* ex. Gadgets/add-ins: building ggplots, regex, and SQL queries then  
  insert the code into source/console editor

The different possible outputs we might want from a Shiny app are:

* To download the RMD or R file as the artifact
* To download a ZIP with source code & data, other supporting files,  
  and the actual rendered result

From there Joe talks about how there are a number of options available  
such as :

1. Copy-paste: Have a Shiny app **and** RMD report
   * Pros: Copy-pasted code is high fidelity and easy to understand
   * Cons: Two copies must be kept in sync and method will not work for  
     more dynamic apps
2. Lexical analysis: automatically generate scripts from app source  
   code (static analysis and heuristics)
   * Pros: Easy to add to app
   * Cons: Not ALL apps can be translated automatically
   * Generated code may **not** be camera ready as it may contain lots of  
     code relating to the Shiny app’s structure
3. Programmatic: Meta-programming techniques to write code for **dual**  
   purposes (execute interactive **and** export static)
   * Pros: Flexible
   * Cons: **High** learning curve and significant effort needed to adapt  
     old Shiny apps

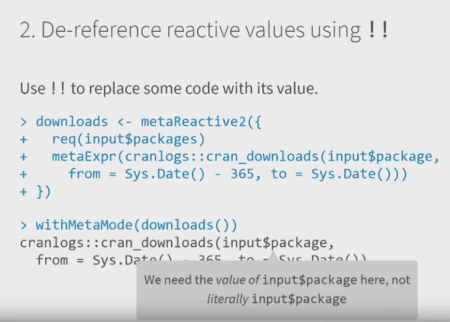
**{shinymeta} package**

There are four main steps to follow when using {shinymeta}:

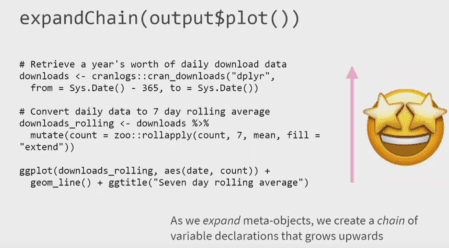
1. Identify the domain logic inside the code and separate it from  
   Shiny’s reactive structure
   * Activate meta mode with withMetaMode() or expandChain()
   * Use metaReactive() to create a reactive() that returns a code  
     expression
   * Other functions to return code include metaObserve(),  
     metaRender(), etc.
   * You can also wrap the code you want with metaExpr() inside  
     function



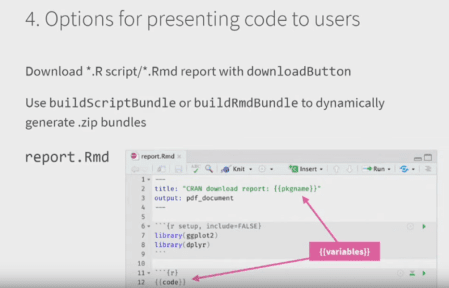
1. Within the domain logic you identified, identify references to  
   reactive values and expressions that need to be replaced with static  
   values and static code
   * De-reference reactive values with !!
   * Replace reactive values with the **actual** values



1. At run time, choose **which** pieces of domain logic to expose to  
   the user
   * expandChain(): turns !! code into variable and introduces code  
     snippet above the function
   * The chain of variable declarations grow upwards as you sequentially  
     expand the meta-objects



1. Present the code to the user!
   * Use outputCodeButton() to add a button for a specific output
   * Use displayCodeModal() to display underlying code
   * Use downloadButton() to allow people to click and download a R  
     script or RMD report
   * Use buildScriptBundle or buidlRmdBundle() to generate .zip  
     bundles dynamically



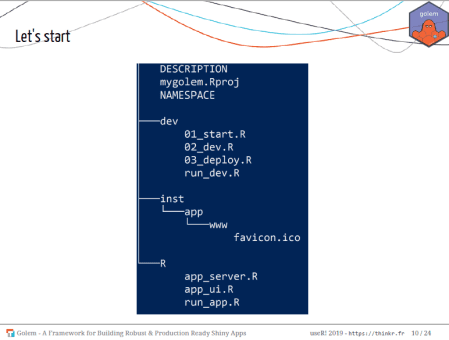
Some of the limitations and future directions Joe, Carson, and the rest  
of the Shiny team acknowledge are that:

* The formatting of the code can be improved (white  
  space not preserved)
* Future compatibility with Shiny async
* So far {shinymeta} only covers reproducing “snapshots” of the app  
  state
* More work and thinking needs to be done to reproduce a “notebook”  
  style record of the how/why/what of the multiple iterations of  
  interactive usage that was needed to get to a certain result and  
  output

**{golem}: Shiny apps in production – Vincent Guyader**

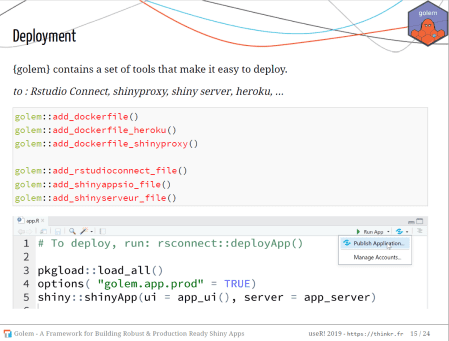
One of the key principles in R is when you are repeatedly writing or  
using the same code or functions then you should write a package, and  
this is no different for Shiny apps as well. The reasons Vincent stated  
were:

* Easy dependency, version, documentation management
* Easy installation and deployment



With the package infrastructure, you need to have the ui.R and  
server.R (app\_ui.R and app\_server.R respectively in {golem}) in  
the R directory and all you need to run your app is the run\_app()  
function.

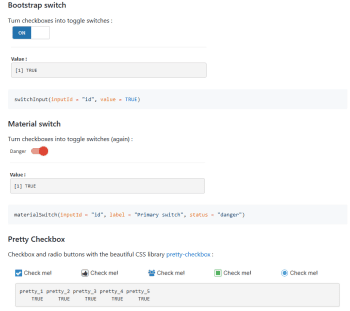
{golem} also has functions that make it easy to deploy your app via R  
Studio Connect, shinyproxy, Shiny server, heroku, etc.



For styling your app with customized JavaScript and CSS files you can  
easily add them to your Shiny app package directory via the  
add\_js\_file() and add\_css\_file() functions. You can do similar but  
with modules with add\_module(). As {golem} is a package you have all  
the great attributes of an R package available to you such as unit  
testing, documentation, and continuous integration/deployment!

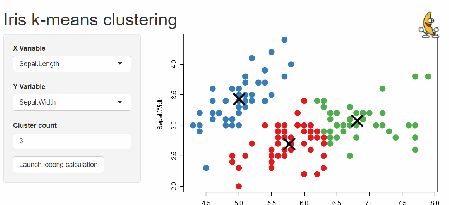
**Our journey with Shiny: Some packages to enhance your applications – Victor Perrier & Fanny Meyer**

The first and probably the most well-known of this group is the  
{shinyWidgets} package which gives you a variety of cool custom widgets that you can add to make your Shiny app via JavaScript and CSS.

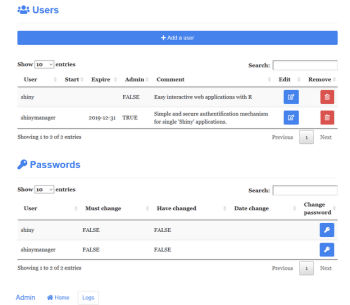


Next, wondering about how exactly users interacted with their Shiny apps  
and whether they used the included widgets the dreamRs team created the  
{shinylogs} package. This packages records any and all inputs that are changed as well as the outputs and errors. This is done by storing the  
JavaScript objects via the  
localForage JavaScript  
library. With this in place shiny developers can see the number of  
connections per day, the user agent family, most viewed tabs, etc.

The {shinybusy} package gives a user feedback when a server operation running or busy such as a spinning circle, a moving bar, or even any  
kind of gif you choose!



Last but not least is the {shinymanager} package which allows you to administrate and manage who can access your application and protects the source code of your app until authentication is successful!



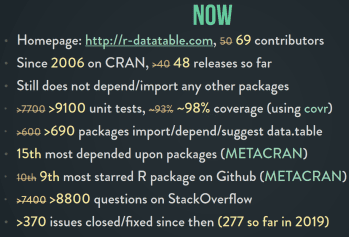
The dreamRs organization are also the organization that created the  
{esquisse} package that lets you interactively make ggplot2 graphs with an RStudio addin!

Talking about packages leads me to the next section…

**Packages**

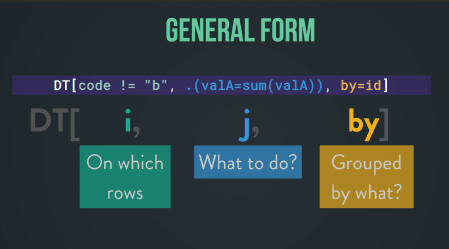
**Summary of developments in R’s data.table package**

Compared to a year ago there has been a lot of change and progress in  
{data.table}:



A key principle of {data.table} is that there are **no** dependencies or  
imports in the package!

The general form of using {data.table} is as follows:



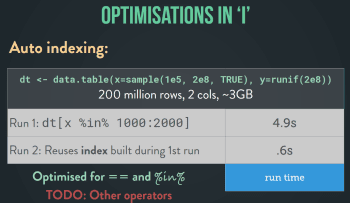
Arun also showed us some examples:

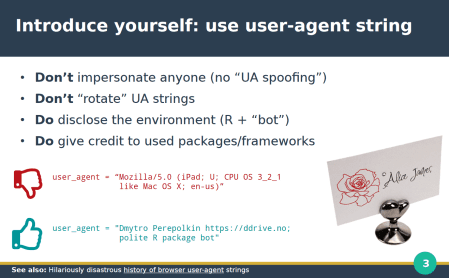

At the end he also talked about the new optimization and functionalities  
in the package.

* for ‘i’: auto-indexing and parallel subsets (columns processed in  
  parallel)
* for ‘j’: using GForce
* for ‘by’: parallelization of radix ordering
* new functionality: froll(), coalesce(), and nafill()



**{polite} – Dmytro Perepolkin**

The {polite} package is one I’ve been used for over a year now (you  
might’ve seen me use it in my soccer or TV data viz) and I was delighted  
to hear that the creator was giving a LT on it!



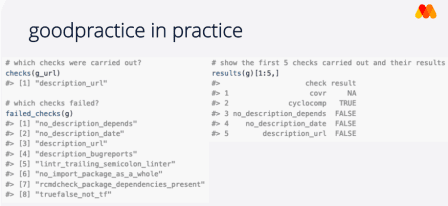
Secondly, you should always check the robots.txt for the website which is a file that  
stipulates various conditions for scraping activity. This can be done  
via {robotstxt} package or by checking the output from polite::bow("theWebsiteYouAreScraping.com")(polite::bow() function  
is what establishes the {polite} session)!

After getting permission you also need to limit the rate at which you  
scrape, you don’t want to overload the servers of the website you are  
using, so **no** parallelization! This can be done with the {ratelimitr}  
package, purrr::slowly() while the {polite} package automatically  
delays by 5 seconds when you run polite::scrape().

After scraping, you should definitely cache your responses with {memoise}, which is what is used inside the polite::scrape() function. Also, wrap your scraper function  
with something like purrr:::safely() so it returns a list of two  
components, a “result” for successes and “error” object for errors in  
your scraping.

**goodpractice: good pkg development Hannah Frick**

By  
running goodpractice::gp() it does static code analysis and can run  
around ~200 of the checks available.



A cool thing you can do is that you can customize the different checks  
it runs, set your own standards beforehand and run the checks based on  
those standards with the make\_check() and make\_prep() functions.  
It’s a great package that I’ve used before at work and for my own  
packages so definitely try it out!