Shiny's Holy Grail

Interactivity with reproducibility

Joe Cheng (@jcheng)

useR! 2019

Shiny: Interactive webapps in R

Shiny: Interactive webapps in R

Shiny: Interactive webapps in R

- Allow users to **quickly explore** different parameter values, dimensions, models/algorithms
- **Faster, more visceral iteration** than modifying/rerunning a traditional R script or R Markdown report/notebook
- Great for **collaboration with domain experts** with no R expertise—no need for direct interaction with R

Demo

jcheng.shinyapps.io

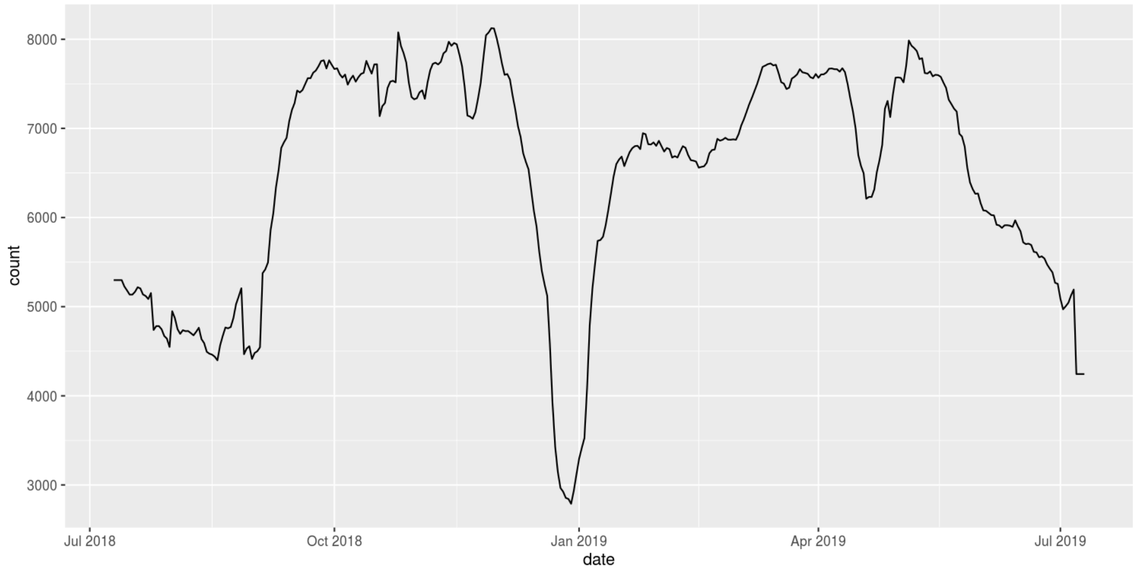
https://jcheng.shinyapps.io/shinymeta-user2019-demo1/

Package name

sp

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    4054    6877    6462    8675   10677
```

Seven day rolling average



jcheng.shinyapps.io

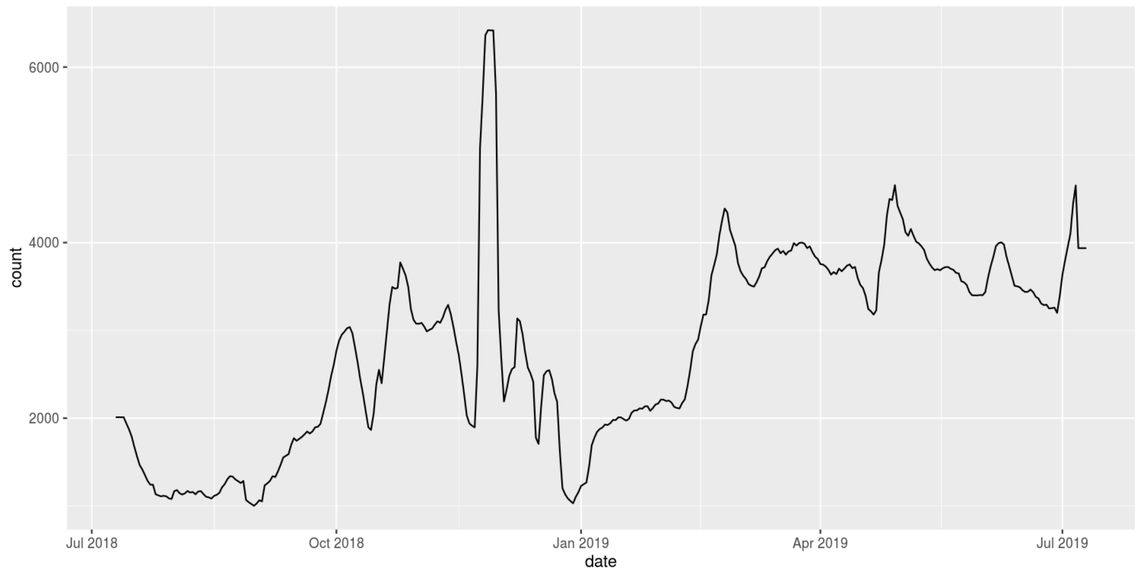
https://jcheng.shinyapps.io/shinymeta-user2019-demo1/

Package name

sf

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    1634    2322    2787    3936   19880
```

Seven day rolling average



But something important is lost

But something important is lost

Interactive apps are powerful and convenient, but reproducibility suffers (vs. R scripts or R Markdown reports)

But something important is lost

Interactive apps are powerful and convenient, but reproducibility suffers (vs. R scripts or R Markdown reports)

- Outputs are transient and not inherently archivable (compared to saving the PDF rendering from a report or script)

But something important is lost

Interactive apps are powerful and convenient, but reproducibility suffers (vs. R scripts or R Markdown reports)

- Outputs are transient and not inherently archivable (compared to saving the PDF rendering from a report or script)
- Reproducing analyses with Shiny is inconvenient: involves not just running the app, but re-enacting the same user interactions

But something important is lost

Interactive apps are powerful and convenient, but reproducibility suffers (vs. R scripts or R Markdown reports)

- Outputs are transient and not inherently archivable (compared to saving the PDF rendering from a report or script)
- Reproducing analyses with Shiny is inconvenient: involves not just running the app, but re-enacting the same user interactions
  - Although bookmarking state is a thing

But something important is lost

Interactive apps are powerful and convenient, but reproducibility suffers (vs. R scripts or R Markdown reports)

- Outputs are transient and not inherently archivable (compared to saving the PDF rendering from a report or script)
- Reproducing analyses with Shiny is inconvenient: involves not just running the app, but re-enacting the same user interactions
  - Although bookmarking state is a thing
- When interactivity is not required or desired, the extra code requirements of Shiny hinder source code clarity

The goal: interactivity + reproducibility

The goal: interactivity + reproducibility

1. First, use interactive app to find interesting results

The goal: interactivity + reproducibility

1. First, use interactive app to find interesting results
2. Then, click a button to view/download a reproducible artifact

The goal: interactivity + reproducibility

1. First, use interactive app to find interesting results
2. Then, click a button to view/download a reproducible artifact

The goal: interactivity + reproducibility

1. First, use interactive app to find interesting results
2. Then, click a button to view/download a reproducible artifact

(You could imagine any number of other ways of bridging interactivity and reproducibility, but I'll be focusing on this specific combination for the rest of the talk.)

The goal: interactivity + reproducibility

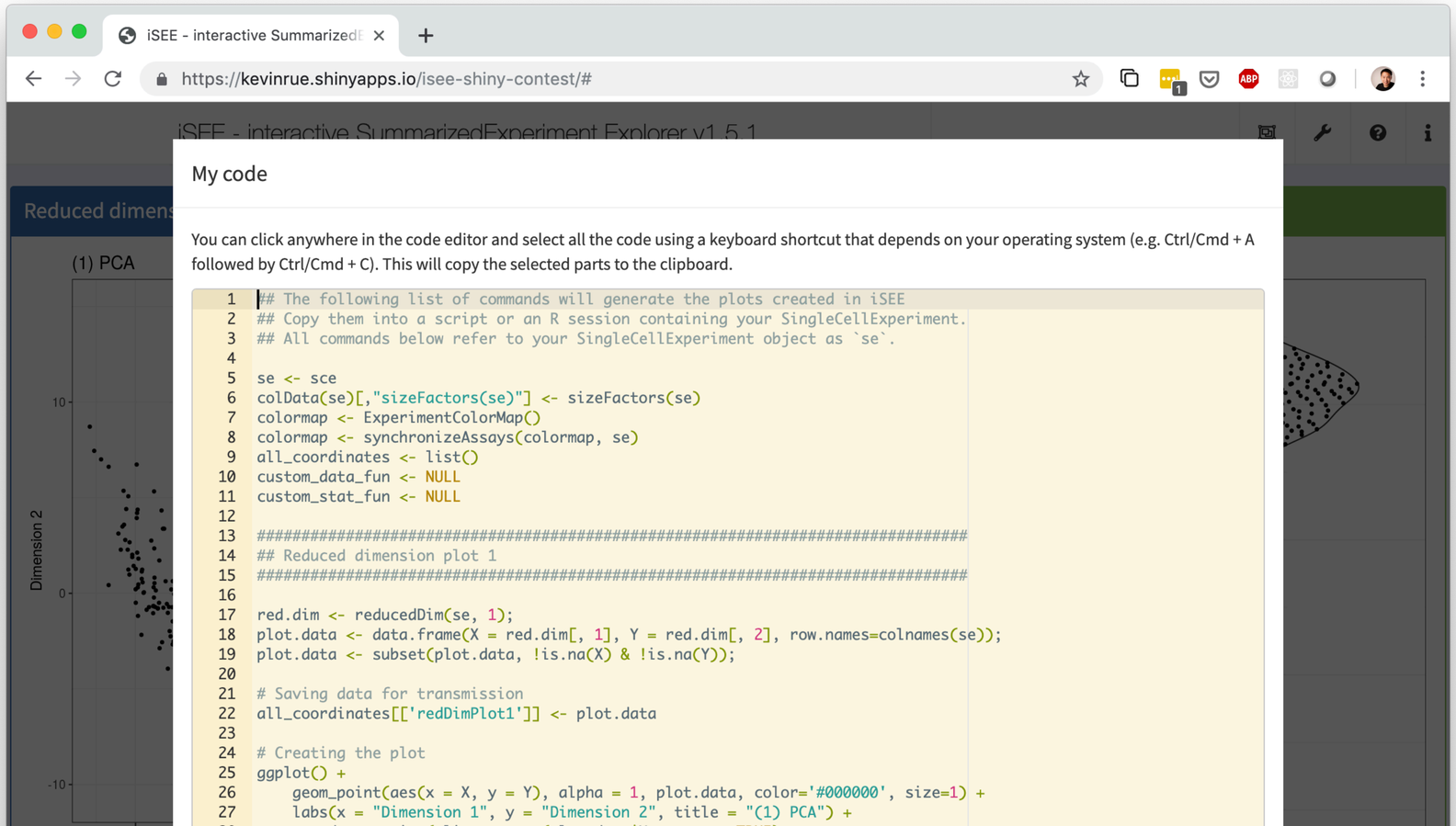
Drug research and validation
Workflows benefit greatly from interactive apps, but analysis ultimately needs to be provided in a fully reproducible form

Teaching
Interactive apps to teach statistical concepts, with corresponding code snippets to teach usage in R

Gadgets/RStudio Add-ins
Use an interactive user interface to build e.g. a ggplot2 plot, regular expression, or SQL query, then insert the corresponding code into the Source editor

Reproducible artifacts

- View an R snippet



iSEE

Reproducible artifacts

- View an R snippet
- Download standalone .Rmd or .R file

Reproducible artifacts

- View an R snippet
- Download standalone .Rmd or .R file
- Download a .zip bundle with source .Rmd/.R, plus…?
  - The outcome of running/rendering the source/script
  - Data files or supporting source code (functions.R)

Demo

jcheng.shinyapps.io

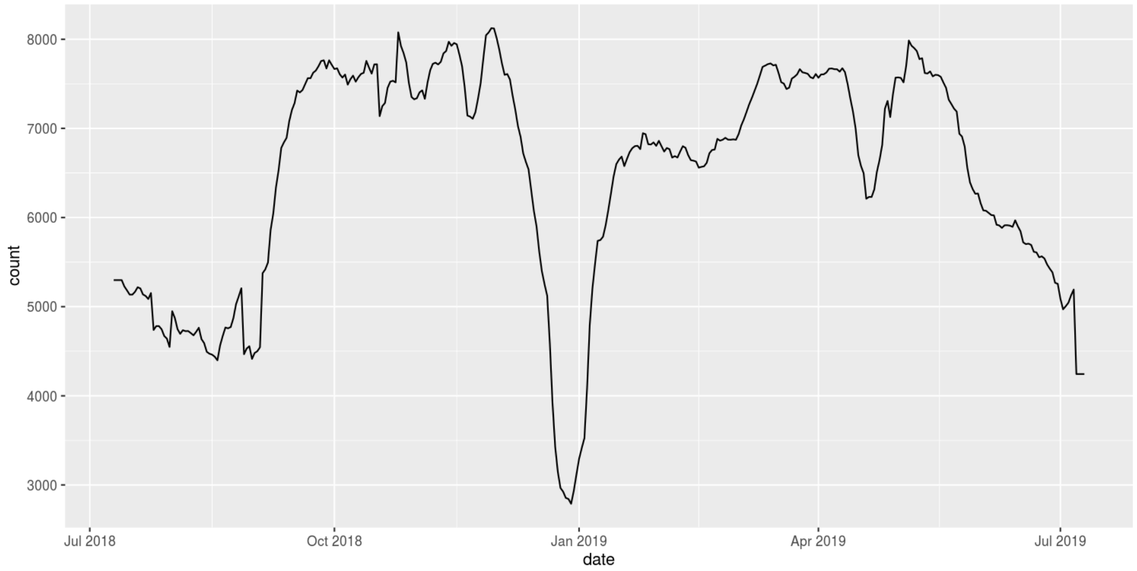
https://jcheng.shinyapps.io/shinymeta-user2019-demo2/

Package name

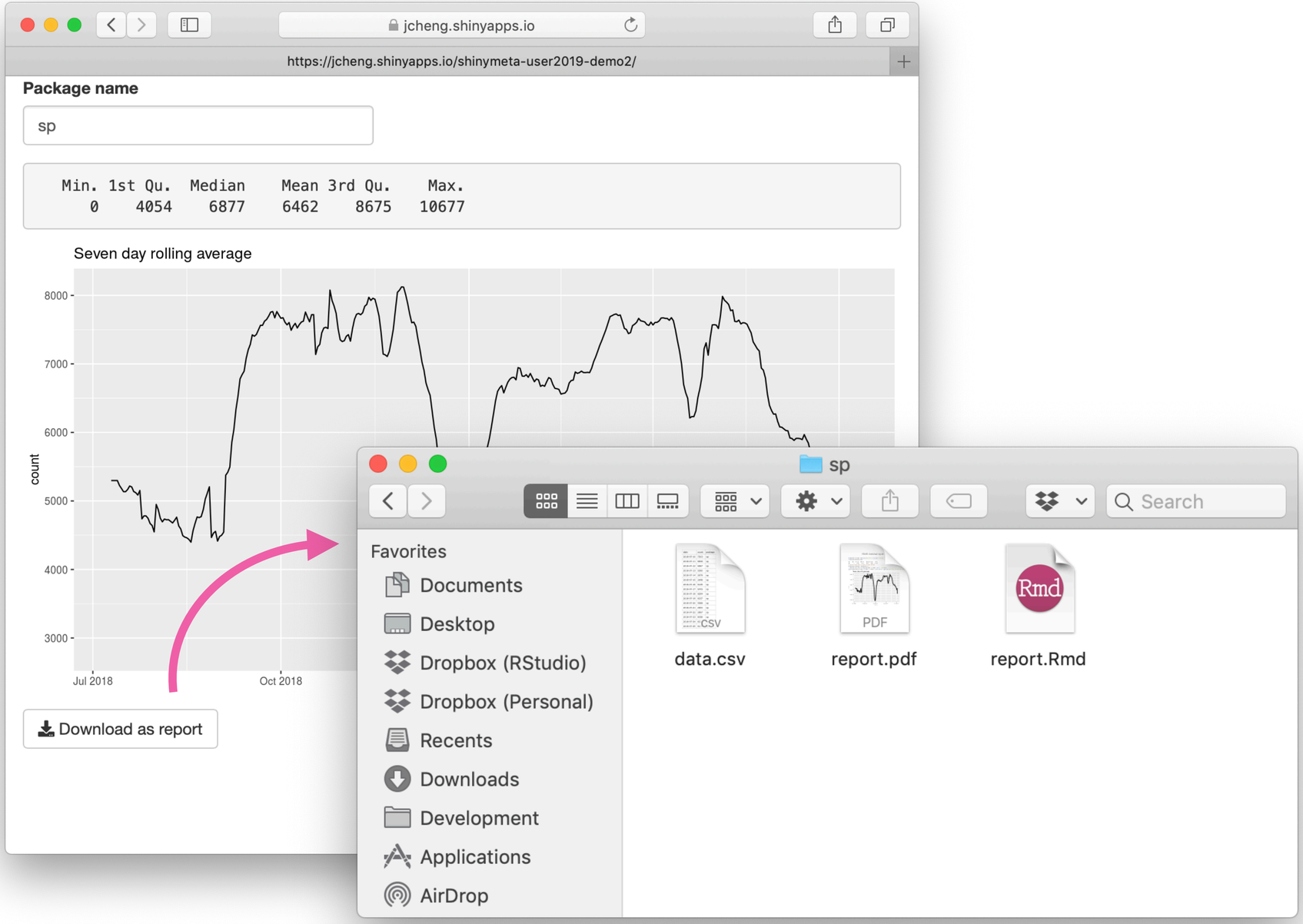
sp

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    4054    6877    6462    8675   10677
```

Seven day rolling average



Download as report


jcheng.shinyapps.io
https://jcheng.shinyapps.io/shinymeta-user2019-demo2/
Package name
sp
Min. 1st Qu. Median Mean 3rd Qu. Max.
0 4054 6877 6462 8675 10677
Seven day rolling average
8000
7000
6000
5000
4000
3000
count
Jul 2018
Oct 2018
Download as report
sp
Search
Favorites
Documents
Desktop
Dropbox (RStudio)
Dropbox (Personal)
Recents
Downloads
Development
Applications
AirDrop
data.csv
report.pdf
report.Rmd

Domain logic vs. reactive structure

Domain logic vs. reactive structure

- **Domain logic** is the essential analysis that our app embodies (loading, data manipulation, modeling, visualization)

Domain logic vs. reactive structure

- **Domain logic** is the essential analysis that our app embodies (loading, data manipulation, modeling, visualization)
- **Reactive structure** is the Shiny-specific server code that makes that analysis interactive

Domain logic vs. reactive structure

- **Domain logic** is the essential analysis that our app embodies (loading, data manipulation, modeling, visualization)
- **Reactive structure** is the Shiny-specific server code that makes that analysis interactive

Shiny app development equals adding reactive structure to your domain logic.

Domain logic vs. reactive structure

- **Domain logic** is the essential analysis that our app embodies (loading, data manipulation, modeling, visualization)
- **Reactive structure** is the Shiny-specific server code that makes that analysis interactive

Shiny app development equals adding reactive structure to your domain logic.

Now we want to take a Shiny app and **extract the domain logic** back out of the reactive structure.

Converting R script to Shiny

```
downloads <- cranlogs::cran_downloads("ggplot2",
  from = Sys.Date() - 365, to = Sys.Date())

downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

Converting R script to Shiny

```
downloads <- reactive({
  cranlogs::cran_downloads("ggplot2",
    from = Sys.Date() - 365, to = Sys.Date())
})

downloads_rolling <- reactive({
  downloads %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- renderPlot({
  ggplot(downloads_rolling, aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Converting R script to Shiny

```
downloads <- reactive({
  cranlogs::cran_downloads(input$packages,
    from = Sys.Date() - 365, to = Sys.Date())
})

downloads_rolling <- reactive({
  downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- renderPlot({
  ggplot(downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Shiny app

```
downloads <- reactive({
  cranlogs::cran_downloads(input$packages,
    from = Sys.Date() - 365, to = Sys.Date())
})

downloads_rolling <- reactive({
  downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- renderPlot({
  ggplot(downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Converting Shiny app to R script

```
downloads <- reactive({
  cranlogs::cran_downloads(input$packages,
    from = Sys.Date() - 365, to = Sys.Date())
})

downloads_rolling <- reactive({
  downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- renderPlot({
  ggplot(downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Converting Shiny app to R script

```
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

R script

```
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

Approach 1: Copy and paste

Create and maintain two separate artifacts: Shiny app and R Markdown report

- 😘 Easy to understand
- 😘 Reproducible code is high fidelity
- 😢 Two copies of code to keep in sync
- 😢 Will not work for more dynamic apps (i.e. not only changing *parameters*, but changing *instructions*)

Approach 2: Lexical analysis

E.g. scriptgloss by Doug Kelkhoff

Automatically generate scripts from app source code, using static analysis and heuristics

- 😘 Very easy to add to your app
- 😘 Few decisions to make (mostly just what outputs are interesting)
- 😢 Not all apps can be translated automatically
- 😢 Generated code is not “camera ready”—still contains code relating only to Shiny structure

Approach 3: Programmatic

Use metaprogramming techniques to write code that serves dual purposes (execute interactively, and export static code)

- 😘 Generated code is almost “camera ready”
- 😘 Flexible enough to handle highly dynamic Shiny apps
- 😢 Higher learning curve
- 😢 Significant effort to adapt existing apps

Introducing shinymeta

by Joe Cheng and Carson Sievert

shinymeta

shinymeta

This package is experimental

- Not tested by QA (yet)
- Function API is still evolving

shinymeta

This package is experimental

- Not tested by QA (yet)
- Function API is still evolving

“Scientists build to learn; Engineers learn to build.” —Fred Brooks

Using shinymeta

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure

2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure

2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively

3. At runtime, **choose which pieces** of domain logic to export, and in what order

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure

2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively

3. At runtime, **choose which pieces** of domain logic to export, and in what order

4. **Present the code** to the user (in a window, as a downloadable script or report, etc.)

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure

2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively

3. At runtime, **choose which pieces** of domain logic to export, and in what order

4. **Present the code** to the user (in a window, as a downloadable script or report, etc.)

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure
2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively
3. At runtime, **choose which pieces** of domain logic to export, and in what order
4. **Present the code** to the user (in a window, as a downloadable script or report, etc.)

1. A new family of reactive objects

1. A new family of reactive objects

What was wrong with Shiny's existing reactive objects?

1. A new family of reactive objects

What was wrong with Shiny's existing reactive objects?

```
downloads <- reactive({
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})
```

1. A new family of reactive objects

What was wrong with Shiny's existing reactive objects?

```
downloads <- reactive({
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})
```

- Call `downloads()` to retrieve the current dataset

1. A new family of reactive objects

What was wrong with Shiny's existing reactive objects?

```
downloads <- reactive({
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})
```

- Call `downloads()` to retrieve the current dataset
- Automatically caches the result until `input$package` changes

1. A new family of reactive objects

What was wrong with Shiny's existing reactive objects?

```
downloads <- reactive({
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})
```

- Call `downloads()` to retrieve the current dataset
- Automatically caches the result until `input$package` changes
- Works well for regular Shiny apps, BUT there's no easy way for us to get the code out

1. A new family of reactive objects

With shinymeta:

```
downloads <- metaReactive({
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})
```

- A `metaReactive` does everything a regular `reactive` does, plus, can give you its own source code at runtime

1. A new family of reactive objects

With shinymeta:

```
downloads <- metaReactive({
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})
```

```
> withMetaMode(downloads())
cranlogs::cran_downloads(input$package, from =
Sys.Date() - 365, to = Sys.Date())
```

1. A new family of reactive objects

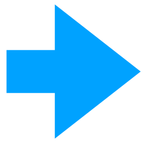
With shinymeta:

All code inside a metaReactive block is considered domain logic

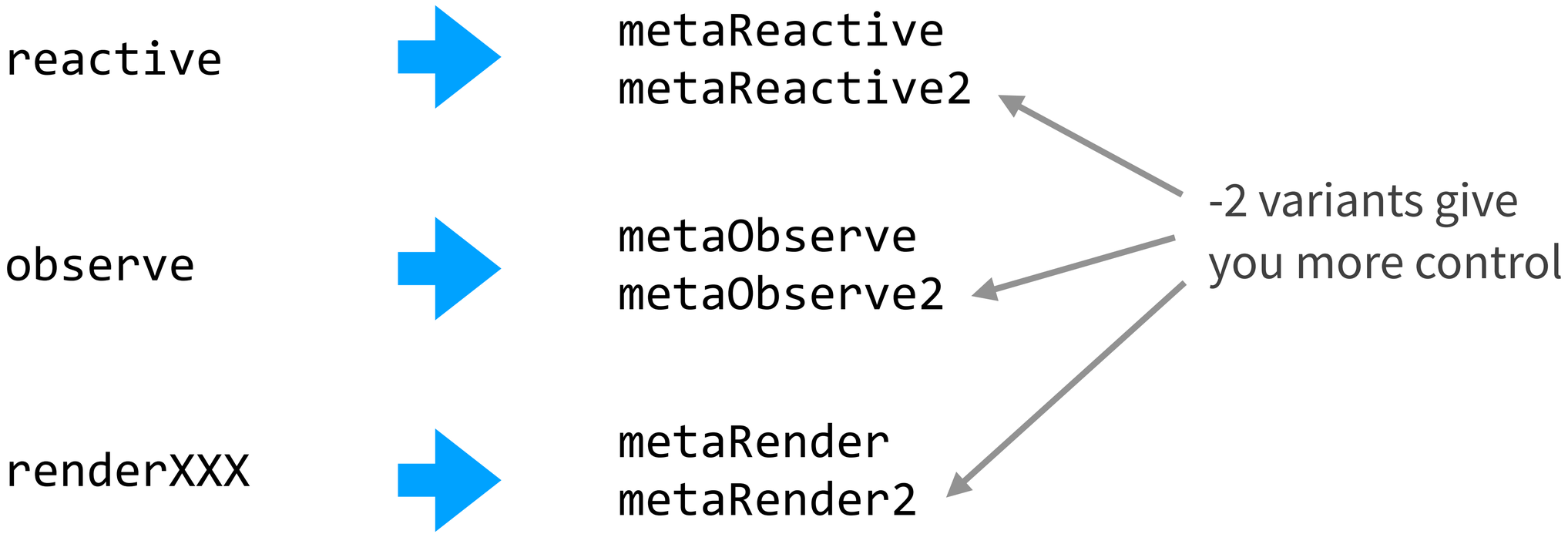
```
downloads <- metaReactive({
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})
```

```
> withMetaMode(downloads())
cranlogs::cran_downloads(input$package, from =
Sys.Date() - 365, to = Sys.Date())
```

1. A new family of reactive objects

`reactive`		`metaReactive` `metaReactive2`
`observe`		`metaObserve` `metaObserve2`
`renderXXX`		`metaRender` `metaRender2`

1. A new family of reactive objects



1. A new family of reactive objects

Sometimes `metaReactive` is too coarse-grained to separate our domain logic from the Shiny stuff:

```
> downloads <- metaReactive({
+   req(input$package)
+   cranlogs::cran_downloads(input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })

> withMetaMode(dataset())
req(input$package)
cranlogs::cran_downloads(input$package,
  from = Sys.Date() - 365, to = Sys.Date())
```

1. A new family of reactive objects

Sometimes `metaReactive` is too coarse-grained to separate our domain logic from the Shiny stuff:

```
> downloads <- metaReactive({
+   req(input$package)
+   cranlogs::cran_downloads(input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })

> withMetaMode(dataset())
req(input$package)
cranlogs::cran_downloads(input$package,
  from = Sys.Date() - 365, to = Sys.Date())
```

Not domain logic

1. A new family of reactive objects

Use `metaReactive2` to tell shinymeta you don't want the entire code chunk, just the part you wrap with `metaExpr()` and return it

```
> downloads <- metaReactive2({
+   req(input$package)
+   metaExpr(cranlogs::cran_downloads(input$package,
+     from = Sys.Date() - 365, to = Sys.Date()))
+ })

> withMetaMode(dataset())
cranlogs::cran_downloads(input$package,
  from = Sys.Date() - 365, to = Sys.Date())
```

Shiny app

```
downloads <- reactive({
  req(input$package)
  cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date())
})

downloads_rolling <- reactive({
  downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- renderPlot({
  ggplot(downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Shinymeta app (…almost)

```
downloads <- metaReactive2({
  req(input$package)
  metaExpr(cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date()))
})

downloads_rolling <- metaReactive({
  downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- metaRender(renderPlot, {
  ggplot(downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Shinymeta app (…almost)

```
downloads <- metaReactive2({
  req(input$package)
  metaExpr(cranlogs::cran_downloads(input$package,
    from = Sys.Date() - 365, to = Sys.Date()))
})

downloads_rolling <- metaReactive({
  downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- metaRender(renderPlot, {
  ggplot(downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure

2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively

3. At runtime, **choose which pieces** of domain logic to export, and in what order

4. **Present the code** to the user (in a window, as a downloadable script or report, etc.)

2. De-reference reactive values using `!!`

Use `!!` to replace some code with its value.

```
> downloads <- metaReactive2({
+   req(input$packages)
+   metaExpr(cranlogs::cran_downloads(input$package,
+     from = Sys.Date() - 365, to = Sys.Date()))
+ })

> withMetaMode(downloads())
cranlogs::cran_downloads(input$package,
  from = Sys.Date() - 365, to = Sys.Date())
```

2. De-reference reactive values using `!!`

Use `!!` to replace some code with its value.

```
> downloads <- metaReactive2({
+   req(input$packages)
+   metaExpr(cranlogs::cran_downloads(input$package,
+     from = Sys.Date() - 365, to = Sys.Date()))
+ })

> withMetaMode(downloads())
cranlogs::cran_downloads(input$package,
  from = Sys.Date() - 365, to = Sys.Date())
```

We need the *value of* `input$package` here, not *literally* `input$package`

2. De-reference reactive values using !!

Use !! to replace some code with its value.


```
> downloads <- metaReactive2({
+   req(input$packages)
+   metaExpr(cranlogs::cran_downloads(!!input$package,
+     from = Sys.Date() - 365, to = Sys.Date()))
+ })

> withMetaMode(downloads())
cranlogs::cran_downloads("ggplot2",
  from = Sys.Date() - 365, to = Sys.Date())
```

2. De-reference reactive exprs using **!!**

2. De-reference reactive exprs using `!!`

Besides inlining *values*, unquoting has a second essential function: inlining *meta-reactive objects* as code

2. De-reference reactive exprs using !!

Besides inlining *values*, unquoting has a second essential function: inlining *meta-reactive objects* as code

```
> downloads <- metaReactive({
+   cranlogs::cran_downloads(!!input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })
```

2. De-reference reactive exprs using !!

Besides inlining *values*, unquoting has a second essential function: inlining *meta-reactive objects* as code

```
> downloads <- metaReactive({
+   cranlogs::cran_downloads(!!input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })

> downloads_rolling <- metaReactive({
+   downloads() %>% mutate(
+     count = zoo::rollapply(count, 7, mean, fill = "extend"))
+ })
```

2. De-reference reactive exprs using !!

Besides inlining *values*, unquoting has a second essential function: inlining *meta-reactive objects* as code

```
> downloads <- metaReactive({
+   cranlogs::cran_downloads(!!input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })

> downloads_rolling <- metaReactive({
+   downloads() %>% mutate(
+     count = zoo::rollapply(count, 7, mean, fill = "extend"))
+ })

> withMetaMode(downloads_rolling())
downloads() %>% mutate(
  count = zoo::rollapply(count, 7, mean, fill = "extend"))
```

2. De-reference reactive exprs using !!

Besides inlining *values*, unquoting has a second essential function: inlining *meta-reactive objects* as code

```
> downloads <- metaReactive({
+   cranlogs::cran_downloads(!!input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })

> downloads_rolling <- metaReactive({
+   !!downloads() %>% mutate(
+     count = zoo::rollapply(count, 7, mean, fill =
"extend"))
+ })

> withMetaMode(downloads_rolling())
downloads() %>% mutate(
  count = zoo::rollapply(count, 7, mean, fill = "extend"))
```

Bang bang!

2. De-reference reactive exprs using !!

Besides inlining *values*, unquoting has a second essential function: inlining *meta-reactive objects* as code

```
> downloads <- metaReactive({
+   cranlogs::cran_downloads(!!input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })

> downloads_rolling <- metaReactive({
+   !!downloads() %>% mutate(
+     count = zoo::rollapply(count, 7, mean, fill =
"extend"))
+ })

> withMetaMode(downloads_rolling())
{
  cranlogs::cran_downloads("ggplot2",
    from = Sys.Date() - 365, to = Sys.Date())
} %>% mutate(
  count = zoo::rollapply(count, 7, mean, fill = "extend"))
```

Bang bang!

Shinymeta app (…almost)

```
downloads <- metaReactive2({
  req(input$packages)
  metaExpr(cranlogs::cran_downloads(input$packages,
    from = Sys.Date() - 365, to = Sys.Date()))
})

downloads_rolling <- metaReactive({
  downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- metaRender(renderPlot, {
  ggplot(downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Shinymeta app

```
downloads <- metaReactive2({
  req(input$packages)
  metaExpr(cranlogs::cran_downloads(!!input$packages,
    from = Sys.Date() - 365, to = Sys.Date()))
})

downloads_rolling <- metaReactive({
  !!downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- metaRender(renderPlot, {
  ggplot(!!downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure
2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively
3. At runtime, **choose which pieces** of domain logic to export, and in what order
4. **Present the code** to the user (in a window, as a downloadable script or report, etc.)

3. Extract code from selected objects

As we've already seen, you can call meta-reactive objects within `withMetaMode()` to extract their code.

```
> downloads <- metaReactive({
+   cranlogs::cran_downloads(!!input$package,
+     from = Sys.Date() - 365, to = Sys.Date())
+ })
> withMetaMode(downloads())
cranlogs::cran_downloads("ggplot2",
  from = Sys.Date() - 365, to = Sys.Date())
```

3. Extract code from selected objects

As we've already seen, you can call meta-reactive objects within `withMetaMode()` to extract their code.

But this was just for demo purposes—in most cases you'll use a smarter, higher-level function called `expandChain()`.

`withMetaMode`

- When `!!` is applied to `input$xxx`, the value is inlined.
- When `!!` is applied to a `metaReactive` read operation, the corresponding code is inlined.

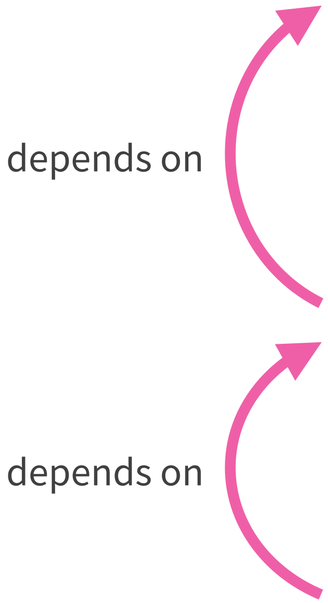
`withMetaMode`

- When `!!` is applied to `input$xxx`, the value is inlined.
- When `!!` is applied to a `metaReactive` read operation, the corresponding code is inlined.

`expandChain`

- When `!!` is applied to `input$xxx`, the value is inlined.
- When `!!` is applied to a `metaReactive` read operation, **a new variable is introduced if one doesn't already exist**.

Example Shinymeta app




```
downloads <- metaReactive2({
  req(input$packages)
  metaExpr(cranlogs::cran_downloads(!!input$packages,
    from = Sys.Date() - 365, to = Sys.Date()))
})

downloads_rolling <- metaReactive({
  !!downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
})

output$plot <- metaRender(renderPlot, {
  ggplot(!!downloads_rolling(), aes(date, count)) +
    geom_line() +
    ggtitle("Seven day rolling average”)
})
```

withMetaMode(output$plot())

```
ggplot(!!downloads_rolling(), aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

withMetaMode(output$plot())

```
ggplot({
  # Convert daily data to 7 day rolling average
  !!downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))
}, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

withMetaMode(output$plot())

```
ggplot({
  # Convert daily data to 7 day rolling average
  !!downloads() %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))
}, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

withMetaMode(output$plot())

```
ggplot({
  # Convert daily data to 7 day rolling average
  {
    # Retrieve a year's worth of daily download data
    cranlogs::cran_downloads("dplyr", from = Sys.Date() -
365, to = Sys.Date())
  } %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))
}, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

withMetaMode(output$plot())

```
ggplot({
  # Convert daily data to 7 day rolling average
  {
    # Retrieve a year's worth of daily download data
    cranlogs::cran_downloads("dplyr", from = Sys.Date() -
365, to = Sys.Date())
  } %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))
}, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
ggplot(!!downloads_rolling(), aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
# Convert daily data to 7 day rolling average
downloads_rolling <- !!downloads() %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
# Convert daily data to 7 day rolling average
downloads_rolling <- !!downloads() %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
# Convert daily data to 7 day rolling average
downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
# Retrieve a year's worth of daily download data
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

# Convert daily data to 7 day rolling average
downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
# Retrieve a year's worth of daily download data
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

# Convert daily data to 7 day rolling average
downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

expandChain(output$plot())

```
# Retrieve a year's worth of daily download data
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

# Convert daily data to 7 day rolling average
downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

As we *expand* meta-objects, we create a *chain* of variable declarations that grows upwards

expandChain(output$plot())

```
# Retrieve a year's worth of daily download data
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

# Convert daily data to 7 day rolling average
downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```

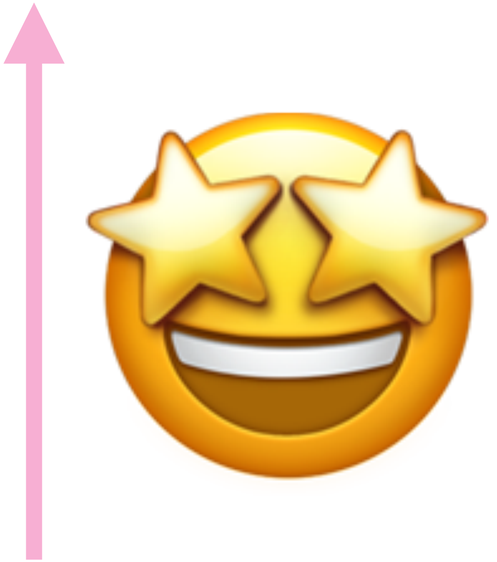
As we *expand* meta-objects, we create a *chain* of variable declarations that grows upwards

expandChain(output$plot())

```
# Retrieve a year's worth of daily download data
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

# Convert daily data to 7 day rolling average
downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")
```



As we *expand* meta-objects, we create a *chain* of variable declarations that grows upwards

Other features of `expandChain`

Other features of `expandChain`

- Can render multiple `meta` objects, by passing multiple arguments

Other features of `expandChain`

- Can render multiple `meta` objects, by passing multiple arguments
- Complex graphs of meta-reactive dependencies are automatically turned into linear code, in the correct order; each dependency object is inserted above the *first* object that needed it

```
withMetaMode(output$plot())
withMetaMode(output$summary())


ggplot({
  # Convert daily data to 7 day rolling average
  {
    # Retrieve a year's worth of daily download data
    cranlogs::cran_downloads("dplyr", from = Sys.Date() - 365, to =
Sys.Date())
  } %>%
    mutate(count = zoo::rollapply(count, 7, mean, fill = "extend"))
}, aes(date, count)) +
  geom_line() +
  ggtitle("Seven day rolling average")

summary({
  # Retrieve a year's worth of daily download data
  cranlogs::cran_downloads("dplyr", from = Sys.Date() - 365, to =
Sys.Date())
}$count)
```

```
expandChain(output$plot(), output$summary())
```

```
# Retrieve a year's worth of daily download data
downloads <- cranlogs::cran_downloads("dplyr",
  from = Sys.Date() - 365, to = Sys.Date())

# Convert daily data to 7 day rolling average
downloads_rolling <- downloads %>%
  mutate(count = zoo::rollapply(count, 7, mean, fill =
"extend"))

ggplot(downloads_rolling, aes(date, count)) +
  geom_line() + ggtitle("Seven day rolling average")

summary(downloads$count)
```

Using shinymeta

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure
2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively
3. At runtime, **choose which pieces** of domain logic to export, and in what order
4. **Present the code** to the user (in a window, as a downloadable script or report, etc.)

4. Options for presenting code to users

Use `outputCodeButton()` to add a button to a *specific* output

```
plotOutput("plot")
```

4. Options for presenting code to users

Use `outputCodeButton()` to add a button to a *specific* output

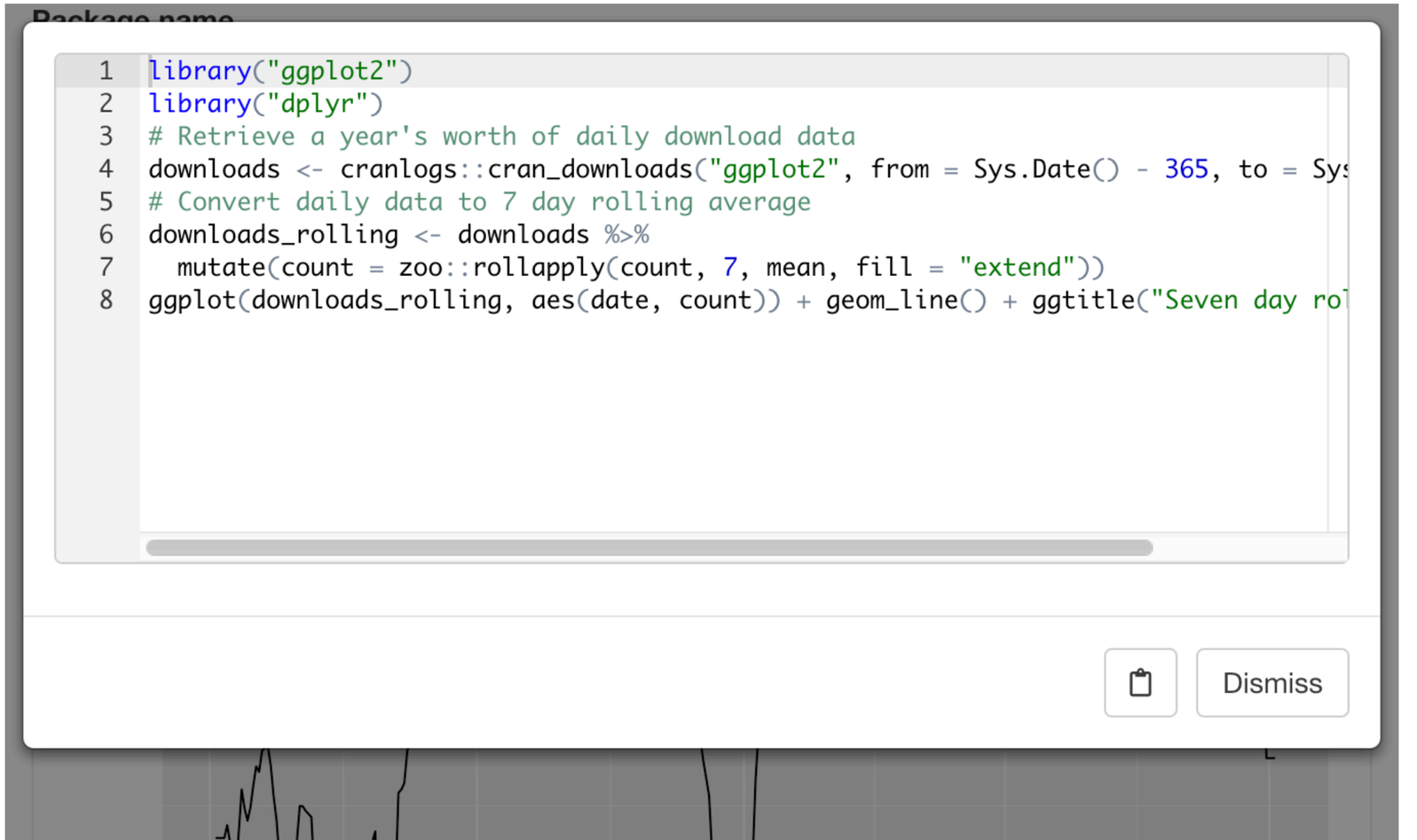
```
outputCodeButton(plotOutput("plot"))
```

4. Options for presenting code to users

Use `outputCodeButton()` to add a button to a *specific* output

4. Options for presenting code to users

Display code using `displayCodeModal()`

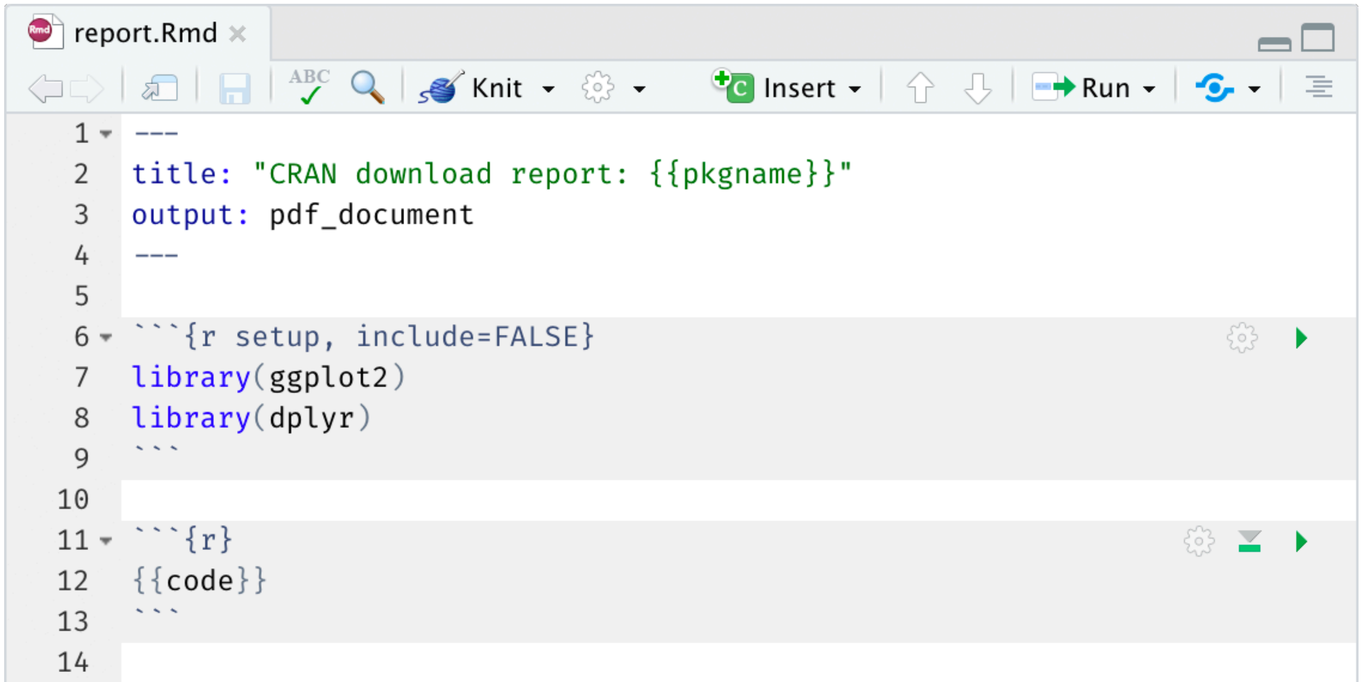


4. Options for presenting code to users

Download *.R script/*.Rmd report with `downloadButton`

Use `buildScriptBundle` or `buildRmdBundle` to dynamically generate .zip bundles

`report.Rmd`

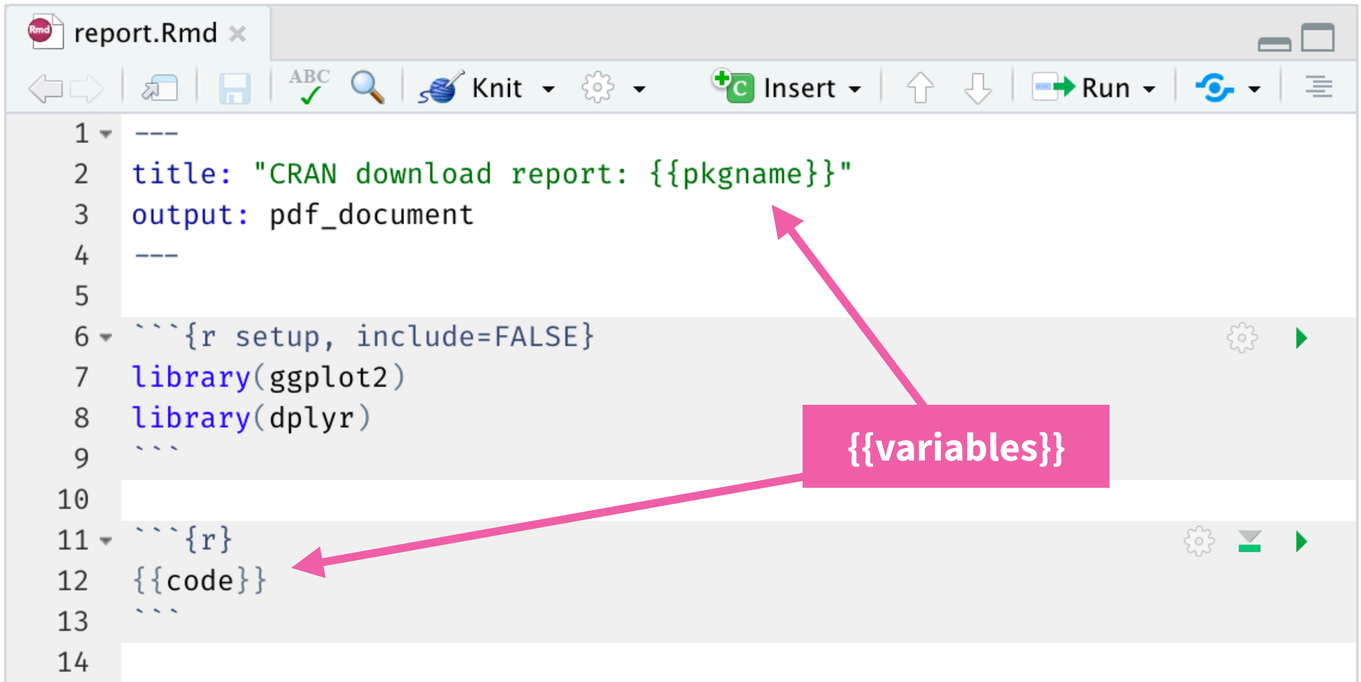


4. Options for presenting code to users

Download *.R script/*.Rmd report with `downloadButton`

Use `buildScriptBundle` or `buildRmdBundle` to dynamically generate .zip bundles

`report.Rmd`



4. Options for presenting code to users

Download *.R script/*.Rmd report with `downloadButton`

Use `buildScriptBundle` or `buildRmdBundle` to dynamically generate .zip bundles

```
buildRmdBundle(
  report_template = "report.Rmd",
  include_files = list("data.csv" = downloads_data),
  vars = list(pkgname = input$package, code = code),
  output_zip_path = out
)
```

Using shinymeta (recap)

1. You (the app author) **identify the domain logic in your app code** so we can separate it from the reactive structure
2. Within that domain logic, you **identify references to reactive values and reactive expressions** that need to be replaced with static values and static code, respectively
3. At runtime, **choose which pieces** of domain logic to export, and in what order
4. **Present the code** to the user (in a window, as a downloadable script or report, etc.)

Limitations and future directions

- Make `expandChain` extract `input`/reactive values as variables
- Formatting of generated code can improve
  - In particular, insignificant whitespace within source code is not preserved
- Compatibility with Shiny async (but should work great with both bookmarking and modules already)
- So far we've only looked at reproducing *snapshots* of app state, not necessarily "lab notebook"-style why/how/what over multiple iterations

Credits

- Special thanks to Adrian Waddell at Roche/Genentech, whose (in-house) teal framework provided direct inspiration for shinymeta
- Thanks to Doug Kelkhoff at Roche/Genentech, whose scriptgloss package provided a valuable counterpoint
- Motivated by functionality built independently by many Shiny users over the years, including Vincent Nijs (radiant); Eric Hare and Andee Kaplan (intRo); Xiao Ni (Novartis); Eric Nantz (Eli Lilly); Kevin Rue, Charlotte Soneson, Federico Marini, and Aaron Lun (iSEE); Tyler Morgan Wall (skpr)

Thank you!

Package docs:
https://rstudio.github.io/shinymeta/

Slides and examples from this talk:
https://github.com/jcheng5/shinymeta-user2019-talk

Appendix: Metaprogramming

What is metaprogramming?

What is metaprogramming?

Writing code that generates/manipulates code

What is metaprogramming?

Writing code that generates/manipulates code

Just like R has built-in objects and functions for working with character data, numeric data, tabular data, etc…

What is metaprogramming?

Writing code that generates/manipulates code

Just like R has built-in objects and functions for working with character data, numeric data, tabular data, etc…

…it also has built-in objects and functions for working with code!

What is metaprogramming?

Writing code that generates/manipulates code

Just like R has built-in objects and functions for working with character data, numeric data, tabular data, etc…

…it also has built-in objects and functions for working with code!

Objects: symbols, calls, expressions

What is metaprogramming?

Writing code that generates/manipulates code

Just like R has built-in objects and functions for working with character data, numeric data, tabular data, etc…

…it also has built-in objects and functions for working with code!

Objects: symbols, calls, expressions

Functions: `quote()`, `as.symbol()`, `call()`, `substitute()`

What is metaprogramming?

Writing code that generates/manipulates code

Just like R has built-in objects and functions for working with character data, numeric data, tabular data, etc…

…it also has built-in objects and functions for working with code!

Objects: symbols, calls, expressions

Functions: `quote()`, `as.symbol()`, `call()`, `substitute()`

Functions in the rlang package: `expr()`, `enexpr()`, `!!`

What is metaprogramming?

Writing code that generates/manipulates code

Just like R has built-in objects and functions for working with character data, numeric data, tabular data, etc…

…it also has built-in objects and functions for working with code!

Objects: symbols, calls, expressions

Functions: `quote()`, `as.symbol()`, `call()`, `substitute()`

Functions in the rlang package: `expr()`, `enexpr()`, `!!`

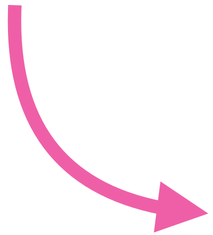
(We won't attempt to cover all this today…)

Creating code objects using quote

```
dplyr::filter(diamonds, carat >= 3)
```

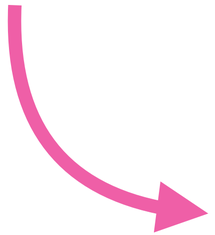
Creating code objects using quote

```
dplyr::filter(diamonds, carat >= 3)
```



Creating code objects using quote

```
dplyr::filter(diamonds, carat >= 3)
```



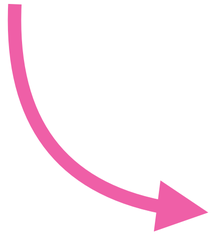
```
# A tibble: 32 x 10
   carat cut     color clarity depth table price     x     y     z
   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  3.01 Premium I     I1       62.7    58  8040  9.1   8.97  5.67
 2  3.11 Fair    J     I1       65.9    57  9823  9.15  9.02  5.98
 3  3.01 Premium F     I1       62.2    56  9925  9.24  9.13  5.73
 4  3.05 Premium E     I1       60.9    58 10453  9.26  9.25  5.66
 5  3.02 Fair    I     I1       65.2    56 10577  9.11  9.02  5.91
 6  3.01 Fair    H     I1       56.1    62 10761  9.54  9.38  5.31
 7  3.65 Fair    H     I1       67.1    53 11668  9.53  9.48  6.38
 8  3.24 Premium H     I1       62.1    58 12300  9.44  9.4   5.85
 9  3.22 Ideal   I     I1       62.6    55 12545  9.49  9.42  5.92
10  3.5  Ideal   H     I1       62.8    57 12587  9.65  9.59  6.03
# … with 22 more rows
```

Creating code objects using quote

```
"dplyr::filter(diamonds, carat >= 3)"
```

Creating code objects using quote

```
"dplyr::filter(diamonds, carat >= 3)"
```



Creating code objects using quote

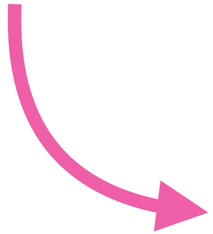
```
"dplyr::filter(diamonds, carat >= 3)"
```

Creating code objects using quote

```
quote(dplyr::filter(diamonds, carat >= 3))
```

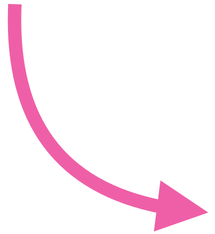
Creating code objects using quote

```
quote(dplyr::filter(diamonds, carat >= 3))
```



Creating code objects using quote

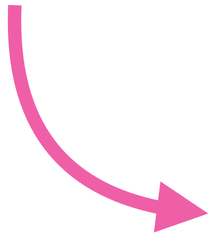
```
quote(dplyr::filter(diamonds, carat >= 3))
```



```
dplyr::filter(diamonds, carat >= 3)
```

Creating code objects using quote

```
quote(dplyr::filter(diamonds, carat >= 3))
```

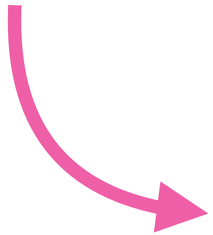


```
\- ()
  \- ()
    \- `::
    \- `dplyr
    \- `filter
  \- `diamonds
  \- ()
    \- `>
    \- `carat
    \-  3
```

(Visualized using `pryr::call_tree()`)

Creating code objects using quote

```
quote(dplyr::filter(diamonds, carat >= 3))
```



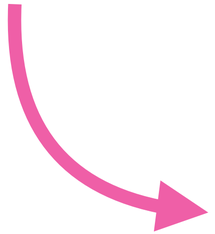
```
\- ()
  \- ()
    \- `::
    \- `dplyr
    \- `filter
  \- `diamonds
  \- ()
    \- `>
    \- `carat
    \-  3
```

(Visualized using `pryr::call_tree()`)

We can read and write nodes within this expression tree, as if it is a list of lists

Creating code objects using ~~quote~~ rlang::expr

```
rlang::expr(dplyr::filter(diamonds, carat >= 3))
```



```
\- ()
  \- ()
    \- `::
    \- `dplyr
    \- `filter
  \- `diamonds
  \- ()
    \- `>
    \- `carat
    \-  3
```

We can read and write nodes within this expression tree, as if it is a list of lists

The unquoting operator (**!!**)

The unquoting operator (`!!`)

```
> min_carat <- 3
```

The unquoting operator (`!!`)

```
> min_carat <- 3

> rlang::expr(dplyr::filter(diamonds, carat >= min_carat))
```

The unquoting operator (`!!`)

```
> min_carat <- 3

> rlang::expr(dplyr::filter(diamonds, carat >= min_carat))
dplyr::filter(diamonds, carat >= min_carat)
```

The unquoting operator (`!!`)

```
> min_carat <- 3

> rlang::expr(dplyr::filter(diamonds, carat >= min_carat))
dplyr::filter(diamonds, carat >= min_carat)

> rlang::expr(dplyr::filter(diamonds, carat >= !!min_carat))
```

The unquoting operator (`!!`)

```
> min_carat <- 3

> rlang::expr(dplyr::filter(diamonds, carat >= min_carat))
dplyr::filter(diamonds, carat >= min_carat)

> rlang::expr(dplyr::filter(diamonds, carat >= !!min_carat))
dplyr::filter(diamonds, carat >= 3)
```

The unquoting operator (`!!`)

```
> min_carat <- 3

> rlang::expr(dplyr::filter(diamonds, carat >= min_carat))
dplyr::filter(diamonds, carat >= min_carat)

> rlang::expr(dplyr::filter(diamonds, carat >= !!min_carat))
dplyr::filter(diamonds, carat >= 3)
```

Use the unquoting operator to selectively replace quoted subexpressions (like `min_carat`) with their **actual values**

The unquoting operator (**!!**)

The unquoting operator (`!!`)

```
> min_carat <- quote(quantile(diamonds$carat, probs = 0.99))
```

The unquoting operator (`!!`)

```
> min_carat <- quote(quantile(diamonds$carat, probs = 0.99))

> rlang::expr(dplyr::filter(diamonds, carat >= !!min_carat))
```

The unquoting operator (`!!`)

```
> min_carat <- quote(quantile(diamonds$carat, probs = 0.99))

> rlang::expr(dplyr::filter(diamonds, carat >= !!min_carat))
dplyr::filter(diamonds, carat >= quantile(diamonds$carat,probs = 0.99))
```

The unquoting operator (**!!**)

```
> min_carat <- quote(quantile(diamonds$carat, probs = 0.99))

> rlang::expr(dplyr::filter(diamonds, carat >= !!min_carat))
dplyr::filter(diamonds, carat >= quantile(diamonds$carat,probs = 0.99))
```

Use the unquoting operator to selectively replace quoted subexpressions (like `min_carat`) with **other quoted expressions**

Further reading

- That's all we'll need to know about metaprogramming for the rest of this talk, but shinymeta works best if you have a solid grasp on the following:
  - *Advanced R* (Wickham) - chapters on Non-standard evaluation and Expressions
  - The dplyr vignette *Programming with dplyr*, specifically the section on quasiquotation

Kelly Bodwin
@KellyBodwin

> stay up all night reading/writing about tidy eval
> drive to work
> this is, zero edits, what the sky looks like

Either I have finally lost my last marble, or the Universe is an #rstats user.

♡ 150 10:22 AM - Jul 2, 2019

24 people are talking about this