Reactive programming

Mine Çetinkaya-Rundel

@minebocek
mine-cetinkaya-rundel
cetinkaya.mine@gmail.com

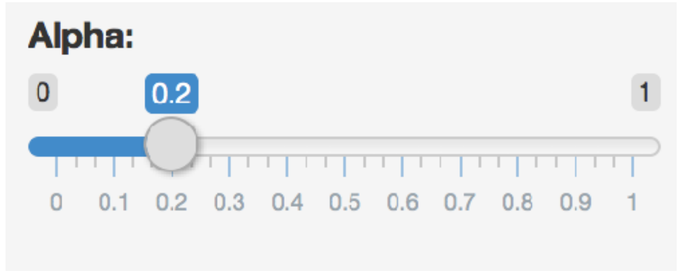
Reactivity 101

Reactions

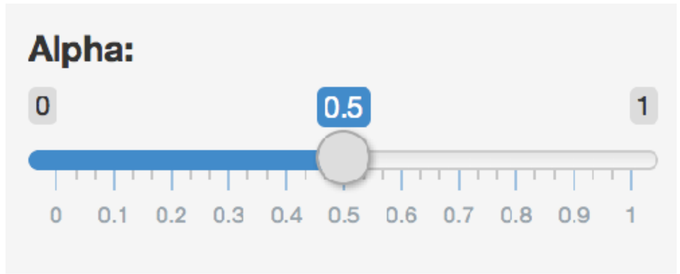
The `input$` list stores the current value of each input object under its name.

```
sliderInput(inputId = "alpha",
            label = "Alpha:",
            min = 0, max = 1,
            value = 0.5)
```

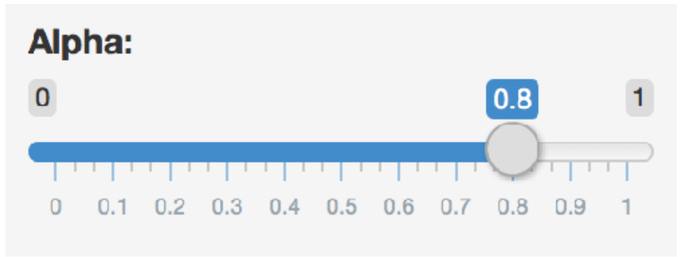
`input$alpha`



`input$alpha = 0.2`



`input$alpha = 0.5`



`input$alpha = 0.8`

Reactivity

Reactivity automatically occurs when an `input` value is used to render an `output` object.

```
01 # Define server function required to create the scatterplot
02 server ← function(input, output) {
03   # Create the scatterplot object the plotOutput function is expecting
04   output$scatterplot ← renderPlot(
05     ggplot(data = movies, aes_string(x = input$x, y = input$y,
06                                      color = input$z)) +
07       geom_point(alpha = input$alpha)
08   )
09 }
```

Your turn

- Modify **`03-react-prog/01-reactivity.R`** to add a new `sliderInput()` defining the size of points (ranging from 0 to 5). Use this variable in the `geom` of the `ggplot` function as the `size` argument. Run the app to ensure that point sizes react when you move the slider.
- **Stretch goal:** Set the interval between each selectable value on the slider to 0.25.

3m 00s

Solution

Solutions to the previous exercises

```
> 03-react-prog/02-reactivity.R
```

Reactivity catalog

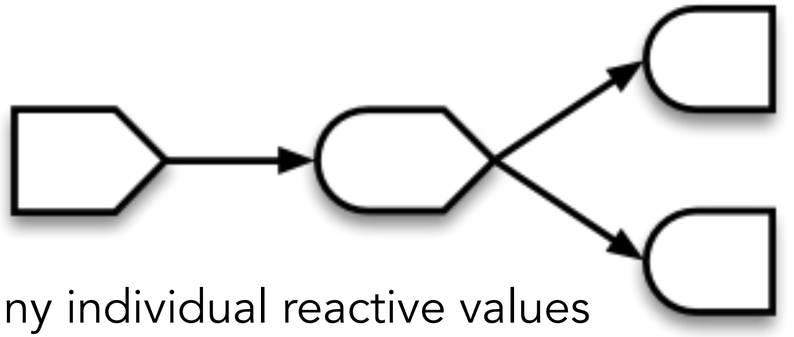
- Store values: `reactiveValues` / `input` / `makeReactiveBinding`
- Calculate values: `reactive` / `eventReactive`
- Execute tasks: `observe` / `observeEvent`
- Preventing reactivity: `isolate`
- Checking preconditions: `req`
- Time (as a reactive source): `invalidateLater`
- Rate-limiting: `debounce` / `throttle`
- Live data: `reactiveFileReader` / `reactivePoll`

Reactivity catalog

- Store values: `reactiveValues` / `input` / `makeReactiveBinding`
- Calculate values: `reactive` / `eventReactive`
- Execute tasks: `observe` / `observeEvent`
- Preventing reactivity: `isolate`
- Checking preconditions: `req`
- Time (as a reactive source): `invalidateLater`
- Rate-limiting: `debounce` / `throttle`
- Live data: `reactiveFileReader` / `reactivePoll`

Highlighted functions are fundamental, all others are built on top.

Implementation of reactives

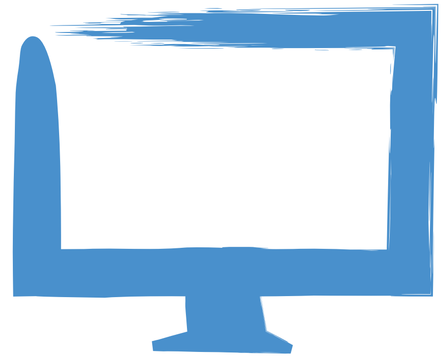


- **Reactive values** – `reactiveValues()`:
  - e.g. `input` is a reactive value, which looks like a list, and contains many individual reactive values that are set by input from the web browser
- **Reactive expressions** – `reactive()`:
  - Can access reactive values or other reactive expressions, and they return a value
  - Useful for caching the results of any procedure that happens in response to user input
- **Observers** – `observe()`:
  - Can access reactive sources and reactive expressions, but they don't return a value; they are used for their side effects
  - e.g. `output` is a reactive observer, which also looks like a list, and contains many individual reactive observers that are created by using reactive values and expressions in reactive functions

Reactive expressions

Reactive expressions

- Open `03-react-prog/03-reactivity.R`, run the app, and observe the new functionality: selecting specific genres of movies.
- Can you spot any inefficiencies in this code? How can we fix it?
- Improved code can be found in `03-react-prog/04-reactivity.R`.



Observers

Observers

- Use to execute actions based on changing reactive values and other reactive expressions.
- Doesn't return a value. So performing side effects is usually the only reason you'd want to create one of these.
- Eagerly executed by Shiny.

Reactive expressions vs. observers

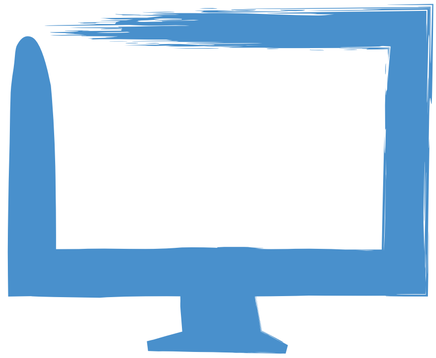
`reactive()`	`observer()`
Callable	Not callable
Returns a value	No return value
Lazy	Eager
Cached	N/A
No side effects	Only for side effects

Reactive expressions vs. observers vs. functions

`reactive()`	`observer()`	`function()`
Callable	Not callable	Callable
Returns a value	No return value	Returns a value
Lazy	Eager	Lazy
Cached	N/A	Not cached
No side effects	Only for side effects	Side effects optional

Observers

- Open **`03-react-prog/05-reactivity.R`**, run the app, and observed the files that get added to the `saved-data` folder. When is a new file written out?
- The behaviour seems a little haphazard. How might you improve it?



Your turn

- Modify **`03-react-prog/05-reactivity.R`** to add a button such that a new file is written out when the button is pressed as opposed to every time `movies_subset()` changes. **Hint:** You will use `observeEvent()` or `eventReactive()`.

5m 00s

Solution

Solutions to the previous exercises

```
> 03-react-prog/06-reactivity.R
```

observeEvent vs. eventReactive

- `observeEvent()` is for event handling
- `eventReactive()` is for delayed computation

```
observeEvent(when_this_changes, {
  do_this
})

r ← eventReactive(when_this_changes, {
  recalculate_this
})
```

Use these functions when you want to **explicitly name your reactive dependencies**, as opposed to letting `reactive/observe` implicitly depend on anything they read.

Your turn

- Open **`03-react-prog/07-reactivity.R`** and run it. This app has several problems:
  - We get an error right off the bat — the plot is running before the user has specified any packages.
  - Unless you're a very fast typist, typing package names will cause the cranlogs server to be queried with many incomplete queries.
- Add an "Update" `actionButton()` to the UI, and make sure nothing happens until it's clicked.

5m 00s

Solution

Solutions to the previous exercises

```
> 03-react-prog/08-reactivity.R
```

Reactive values

Reactive values

- Reactive values are read/write versions of `input`.
- `reactiveValues()` returns an object for storing reactive values — similar to a list, but…
  - when you read a value from it, the calling reactive expression takes a reactive dependency on that value, and
  - when you write to it, it notifies any reactive functions that depend on that value.

```
# Create
rv ← reactiveValues(x = 10)

# Read
rv$x

# Write
rv$x ← 20
```

Your turn

- Open **`03-react-prog/09-reactivity.R`** and run it. It has three action buttons:
  - Increment: Increase the value by 1
  - Decrement: Decrease the value by 1
  - Reset: Set the value to 0
- Unfortunately, it doesn't work.
- Implement the server side. **Hint:** Use `reactiveValues()`!

5m 00s

Solution

Solutions to the previous exercises

```
> 03-react-prog/10-reactivity.R
```

Tip

- **Don't** use `reactiveValues()` when you're calculating a value based on other values and calculations that are already available to you.

- **Do** use `reactiveValues()` to store state that otherwise would be lost from your graph of reactive objects.

Preventing reactivity

Preventing reactivity

- Use `isolate()` from inside a reactive expression or observer, to ignore the implicit reactivity of a piece of code.
- Wrap it around expressions or a whole code block.

Question

Determine when `r1`, `r2`, and `r3` update.

```
r1 ← reactive({
  input$x * input$y
})

r2 ← reactive({
  input$x * isolate({ input$y })
})

r3 ← reactive({
  isolate({ input$x * input$y })
})
```

Solution

```
# Updates every time input$x or input$y change
r1 ← reactive({
  input$x * input$y
})

# Updates only when input$x changes
r2 ← reactive({
  input$x * isolate({ input$y })
})

# Never updates; it will always have its original value
r3 ← reactive({
  isolate({ input$x * input$y })
})
```

Checking preconditions

Checking preconditions

- Cancel the current output (or observer) if a condition isn't met.
  - `req(input$text)`: Ensure the user has provided a value for the "text" input
  - `req(input$button)`: Ensure the button has been pressed at least once
  - `req(x %% 2 == 0)`: Ensure that x is an even number
  - `req(FALSE)`: Unconditionally cancel the current reactive, observer, or output

Checking preconditions

- `req(cond)` is similar to:
  - `stopifnot(cond)`
  - `if (!cond) stop()`
  - `assertthat::assert_that(cond)`
- But with these differences:
  - Errors during output rendering show up with bold red text in the UI; `req` just makes the output blank.
  - Rather than verifying that `cond` is true, req verifies that `cond` is *truthy* (see ?isTruthy)
    - Feels unnatural to be so arbitrary and nebulous, but this definition is just too practical for UI programming.
  - Most importantly, `req` is like an error in that it "infects" the downstream elements of the reactive graph.

Your turn

- Open **`03-react-prog/11-reactivity.R`** and run it. It has lots of errors in the browser and the R console — ignore those for the moment.
- From the app, upload the `diamonds.csv` file found in the same directory. Now everything looks good.
- Diagnose why the errors appear when the app first comes up, and how you can get them to go away. **Hint:** Use `req()`.

5m 00s

Solution

Solutions to the previous exercises

```
> 03-react-prog/12-reactivity.R
```

Time as a reactive source

Question

What will this produce?

```
01 ui ← basicPage(
02   verbatimTextOutput("text")
03   )
04
05 server ← function(input, output){
06
07   r ← reactive({ Sys.time() })
08   output$text ← renderPrint({ r() })
09
10 }
11
12 shinyApp(ui, server)
```

Solution

```
01 ui ← basicPage(
02   verbatimTextOutput("text")
03   )
04
05 server ← function(input, output){
06
07   r ← reactive({ Sys.time() })
08   output$text ← renderPrint({ r() })
09
10 }
11
12 shinyApp(ui, server)
```

An app that reports `Sys.time()` at the time of first launch, and then doesn't update it.

Question

What will this produce?

```
01 ui ← basicPage(
02   verbatimTextOutput("text")
03   )
04
05 server ← function(input, output){
06
07   r ← reactive({
08     invalidateLater(1000)
09     Sys.time()
10   })
11   output$text ← renderPrint({ r() })
12
13 }
14
15 shinyApp(ui, server)
```

Solution

```
01 ui ← basicPage(
02   verbatimTextOutput("text")
03   )
04
05 server ← function(input, output){
06
07   r ← reactive({
08     invalidateLater(1000)
09     Sys.time()
10   })
11   output$text ← renderPrint({ r() })
12
13 }
14
15 shinyApp(ui, server)
```

An app updates reported `Sys.time()` every second.

Limiting rate

Debounce and throttle

- If a reactive value or expression changes too fast for downstream calculations to keep up, your users will have a bad experience (laggy experience, wasted work).
  - `debounce()` and `throttle()` take a reactive expression object as input, and return a rate-limited version of that reactive expression.

```
# A reactive that updates as often as every 50 milliseconds
fast_reactive ← reactive({ ... })

# A reactive that updates no more often than every 2000 milliseconds
throttled_reactive ← throttle(fast_reactive, 2000)

# A reactive that doesn't update until fast_reactive has stopped
# changing for at least 1000 milliseconds
debounced_reactive ← debounce(fast_reactive, 1000)
```

Your turn

- Open **`03-react-prog/13-reactivity.R`** and run it. Click on the plot a few times to create points. Notice the annoying laggy behaviour — this is due to a (simulated) expensive summary output.
- Use `debounce()` or `throttle()` to prevent the summary output from running so often.

5m 00s

Solution

Solutions to the previous exercises

```
> 03-react-prog/14-reactivity.R
```

Tip

- This is not a true debounce/throttle in that it will not prevent R from being called many times, but rather, the reactive invalidation signal that is produced by R is debounced/throttled instead.

- These functions should be used when R is cheap but the things it will trigger (downstream outputs and reactives) are expensive.

Reactive programming

Mine Çetinkaya-Rundel

@minebocek
mine-cetinkaya-rundel
cetinkaya.mine@gmail.com