250 Northern Ave, Boston, MA 02210
Phone: 844-448-1212

Email: info@rstudio.com
Web: http://www.rstudio.com

Writing Big Apps

How to use Shiny Modules

Garrett Grolemund and Joe Cheng

January 2016

Follow @rstudio

HELLO

my name is

Garrett

@StatGarrett

Warm Ups

Choose a partner

```
ui <- fluidPage(
  sliderInput("slider", "Slide Me", 0, 100, 1),
  textOutput("num")
)

server <- function(input, output) {
  output$num <- renderText({
    input$slider
  })
}

shinyApp(ui, server)
```

Will this work?

```r
slider <- function() {
  sliderInput("slider", "Slide Me", 0, 100, 1)
}
ui <- fluidPage(
  slider(),
  textOutput("num")
)
server <- function(input, output) {
  output$num <- renderText({input$slider})
}
shinyApp(ui, server)
```

Will this work?

```
slider <- function() {
  sliderInput("slider", "Slide Me", 0, 100, 1)
}
ui <- fluidPage(
  slider(),
  textOutput("num1"),
  slider(),
  textOutput("num2")
)
server <- function(input, output) {
  output$num1 <- renderText({input$slider})
  output$num2 <- renderText({input$slider})
}
shinyApp(ui, server)
```

Will this work?

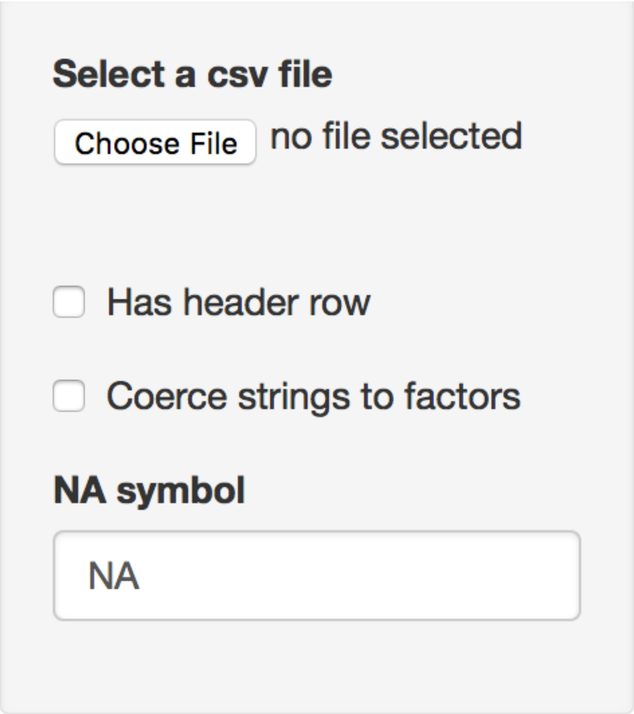
```
slider <- function(id) {
  sliderInput(id, "Slide Me", 0, 100, 1)
}
ui <- fluidPage(
  slider("slider1"),
  textOutput("num1"),
  slider("slider2"),
  textOutput("num2")
)
server <- function(input, output) {
  output$num1 <- renderText({input$slider1})
  output$num2 <- renderText({input$slider2})
}
shinyApp(ui, server)
```

Will this work?

Shiny Modules

What is a module?

A self-contained, composable component of a Shiny App.



Why use modules?

Reuse

Quickly reuse the same code in different apps, or multiple times in the same app.

Isolate

Divide code into separate modules that can be reasoned about independently.

Why use modules?

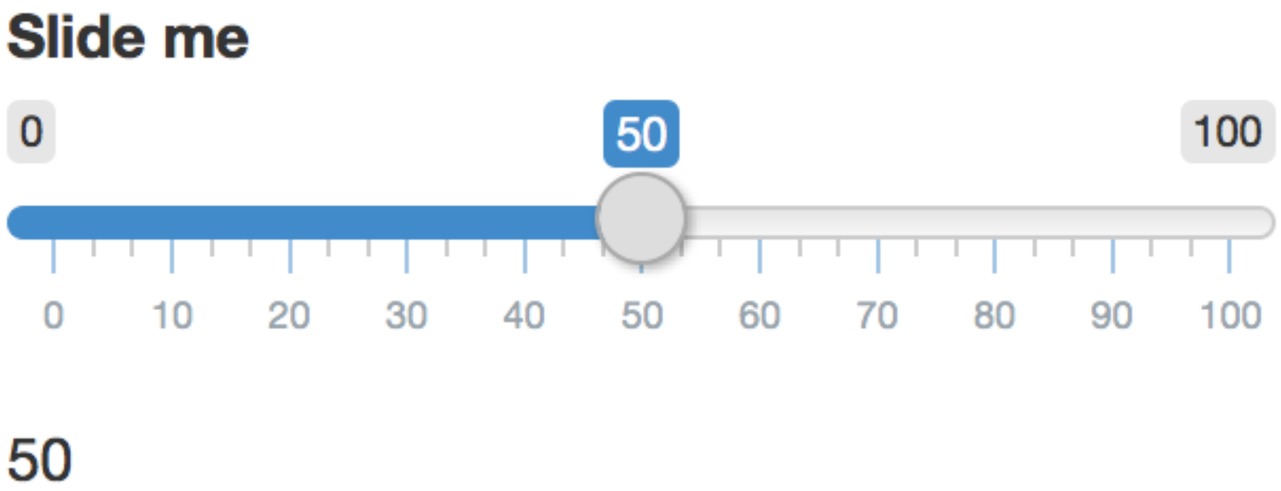
DEMO

Modules Pattern

What is a module?

A pattern of code

- A function that creates UI elements
- A function that loads server logic




```
sliderTextUI <- function(){ #UI }
sliderText <- function(){ #Server logic }
```

Naming practice

A shared root that describes the module

Suffix the ui function with UI, Input, or Output

```
sliderTextUI <- function(){ #UI }
sliderText <- function(){ #Server logic }
```

Naming practice

A shared root that describes the module

Suffix the ui function with UI, Input, or Output

```
sliderTextUI <- function(){ #UI }
sliderText <- function(){ #Server logic }
```

Module UI

Task 1 - Return Shiny UI. Wrap multiple elements in `tagList()`

```
sliderTextUI <- function(){

  tagList(
    sliderInput("slider", "Slide me", 0, 100, 1),
    textOutput("number")
  )
}
```

Module UI

Task 2 - Assign module elements to a unique namespace with NS()

```
sliderTextUI <- function(){

  tagList(
    sliderInput("slider", "Slide me", 0, 100, 1),
    textOutput("number")
  )
}
```

namespaces

A system for organizing objects with identical names.

For example, R's package namespace system.

NS( )

Available in Shiny >= 0.13

```
NS("hello")
## function (id)
## {
##     paste(c(namespace, id), collapse = ns.sep)
## }
## <environment: 0x1034976e8>
```

Module UI

Task 2 - Assign module elements to a unique namespace with NS()

```
sliderTextUI <- function(id){
  ns <- NS(id)
  tagList(
    sliderInput(ns("slider"), "Slide me", 0, 100, 100, 1),
    textOutput(ns("number"))
  )
}
```

1. Add an id argument
2. Make a namespace function
3. Wrap all inputId's and outputId's

How to use Module UI

Call the module UI function where you want the UI elements to go. Supply a unique ID.

```
ui <- fluidPage(
  sliderTextUI("one")
)

server <- function(input, output) {}

shinyApp(ui, server)
```

Module server

Handles the server logic for the module. Very similar to a Shiny app server function.

```
sliderText <- function(input, output, session){
  output$number <- renderText({
    input$slider
  })
}
```

Module server

Handles the server logic for the module. Very similar to a Shiny app server function.

❶ You must use all 3 arguments: input, output, session

```
sliderText <- function(input, output, session){
  output$number <- renderText({
    input$slider
  })
}
```

Module server

Handles the server logic for the module. Very similar to a Shiny app server function.

```
sliderText <- function(input, output, session){
  output$number <- renderText({
    input$slider
  })
}
```

1. You must use all 3 arguments: input, output, session
2. Refer to inputs and outputs from the module without ns()

How to use Module Server

Load the module server function in your app's server function with `callModule()`

```
ui <- fluidPage(
  sliderTextUI("one")
)
server <- function(input, output) {
  callModule(sliderText, "one")
}
shinyApp(ui, server)
```

How to use Module Server

Load the module server function in your app's server function with `callModule()`

```
ui <- fluidPage(
  sliderTextUI("one")
)
server <- function(input, output) {
  callModule(sliderText, "one")
}
shinyApp(ui, server)
```

❶ Call from within server function

How to use Module Server

Load the module server function in your app's server function with `callModule()`

```
ui <- fluidPage(
  sliderTextUI("one")
)
server <- function(input, output) {
  callModule(sliderText, "one")
}
shinyApp(ui, server)
```

1. Call from within server function
2. 1st argument = module function

How to use Module Server

Load the module server function in your app's server function with `callModule()`

```
ui <- fluidPage(
  sliderTextUI("one")
)
server <- function(input, output) {
  callModule(sliderText, "one")
}
shinyApp(ui, server)
```

1. Call from within server function
2. 1st argument = module function
3. 2nd argument = same id as UI

Where to define the module functions?

1. In the preamble of a single file app (app.R)
2. In a file that is sourced in the preamble of a single file app (app.R)
3. In global.R
4. In a file sourced by global.R
5. In a package that the app loads

Your Turn

Open Shiny-Modules/Exercise-1, which contains a simple version of the gapMinder app.

Refactor the app into a module.

Then call the module from an app to recreate the simple version of the gapMinder app.

```
gapModuleUI <- function(id) {
  ns <- NS(id)

  tagList(
    plotOutput(ns("plot")),
    sliderInput(ns("year"), "Select Year",
      value = 1952, min = 1952,
      max = 2007, step = 5,
      animate = animationOptions(interval = 500))
  )
}
```

```
gapModule <- function(input, output, session) {
  ydata <- reactive({ filter(gapminder, year == input$year) })
  xrange <- range(gapminder$gdpPercap)
  yrange <- range(gapminder$lifeExp)

  output$plot <- renderPlot({
    plot(gapminder$gdpPercap, gapminder$lifeExp, type = "n", xlab = "GDP per capita",
      ylab = "Life Expectancy", panel.first = {
         grid()
         text(mean(xrange), mean(yrange), input$year, col = "grey90", cex = 5)
       })

    legend("bottomright", legend = levels(gapminder$continent), cex = 1.3, inset = 0.01,
      text.width = diff(xrange)/5, fill = c("#E41A1C99", "#377EB899", "#4DAF4A99",
      "#984EA399", "#FF7F0099"))

    cols <- c("Africa" = "#E41A1C99", "Americas" = "#377EB899", "Asia" = "#4DAF4A99",
      "Europe" = "#984EA399", "Oceania" = "#FF7F0099")[ydata()$continent]

    symbols(ydata()$gdpPercap, ydata()$lifeExp, circles = sqrt(ydata()$pop), bg = cols,
      inches = 0.5, fg = "white", add = TRUE)
  })
}
```

```r
gapModule <- function(input, output, session) {
  ydata <- reactive({ filter(gapminder, year == input$year) })
  xrange <- range(gapminder$gdpPercap)
  yrange <- range(gapminder$lifeExp)

  output$plot <- renderPlot({
    plot(gapminder$gdpPercap, gapminder$lifeExp, type = "n", xlab = "GDP per capita",
      ylab = "Life Expectancy", panel.first = {
         grid()
         text(mean(xrange), mean(yrange), input$year, col = "grey90", cex = 5)
       })

    legend("bottomright", legend = levels(gapminder$continent), cex = 1.3, inset = 0.01,
      text.width = diff(xrange)/5, fill = c("#E41A1C99", "#377EB899", "#4DAF4A99",
      "#984EA399", "#FF7F0099"))

    cols <- c("Africa" = "#E41A1C99", "Americas" = "#377EB899", "Asia" = "#4DAF4A99",
      "Europe" = "#984EA399", "Oceania" = "#FF7F0099")[ydata()$continent]

    symbols(ydata()$gdpPercap, ydata()$lifeExp, circles = sqrt(ydata()$pop), bg = cols,
      inches = 0.5, fg = "white", add = TRUE)
  })
}
```

```r
library(shiny)
library(gapminder)
library(dplyr)
source("gapModule.R")

ui <- fluidPage(
  gapModuleUI("all")
)

server <- function(input, output) {
  callModule(gapModule, "all")
}

shinyApp(ui = ui, server = server)
```

```r
library(shiny)
library(gapminder)
library(dplyr)
source("gapModule.R")

ui <- fluidPage(
  gapModuleUI("all")
)

server <- function(input, output) {
  callModule(gapModule, "all")
}

shinyApp(ui = ui, server = server)
```

Re-use
Modules

Arguments

Module functions are functions: you can add and use extra arguments.

```
sliderTextUI <- function(id, label = "Slide me"){
  ns <- NS(id)

  tagList(
    sliderInput(ns("slider"), label, 0, 100, 1),
    textOutput(ns("number"))
  )
}
```

Reusing modules

Give the module a unique id each time you call it.

```
ui <- fluidPage(
  sliderTextUI("first", label = "Choose a number"),
  sliderTextUI("second", label = "Choose a number"),
  sliderTextUI("third", label = "Choose a number")
)
server <- function(input, output) {
  callModule(sliderText, "first")
  callModule(sliderText, "second")
  callModule(sliderText, "third")
}
shinyApp(ui, server)
```

Your Turn

Open Shiny-Modules/Exercise-2, which contains a skeleton of the complete gapMinder app.

Modify the module in gapMinder.R to accept a data frame as an argument.

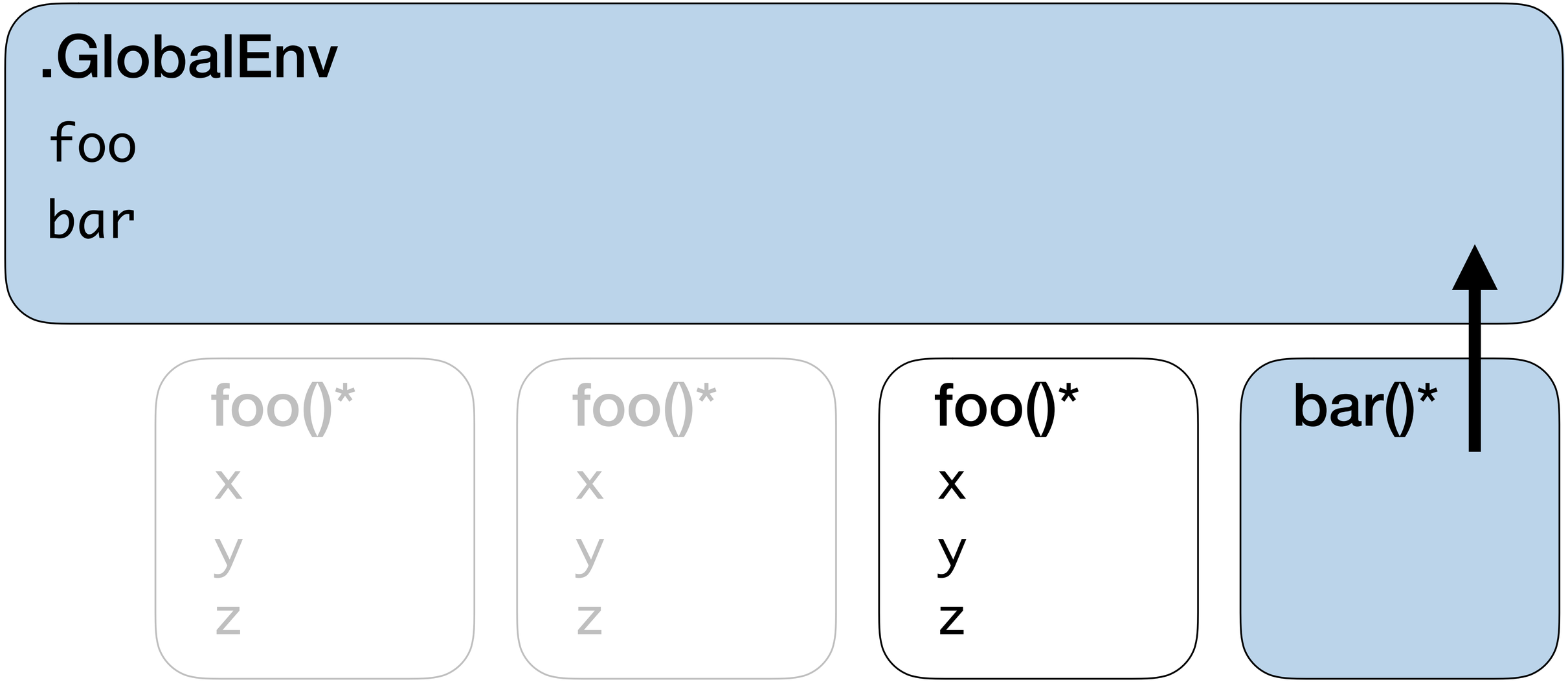
Then complete the app skeleton in app.R by calling the module multiple times.

ReWarm

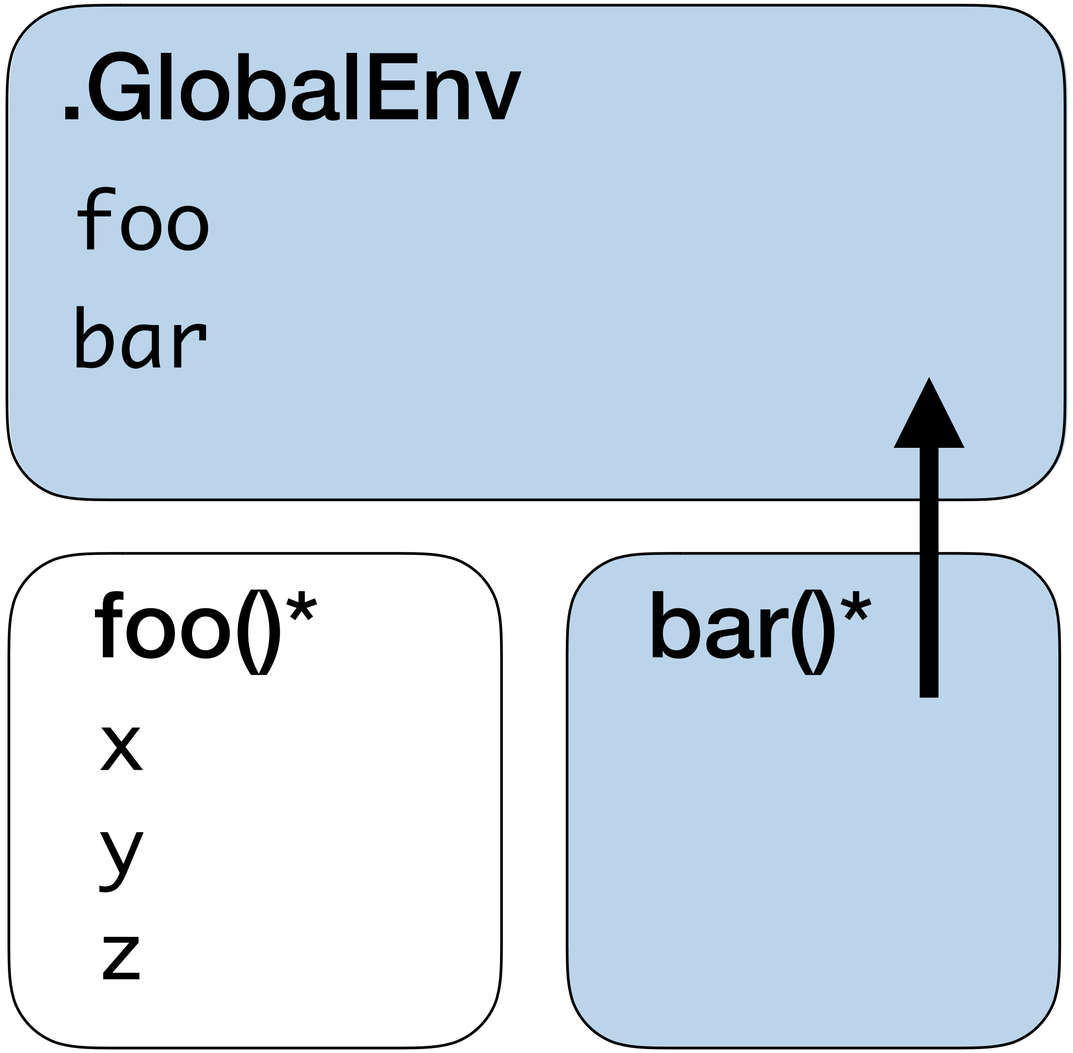
Choose a partner

```
foo <- function() {
  x <- 1
  y <- 2
  z <- 3
}

bar <- function() {
  x + 1
}

foo()
bar()
```

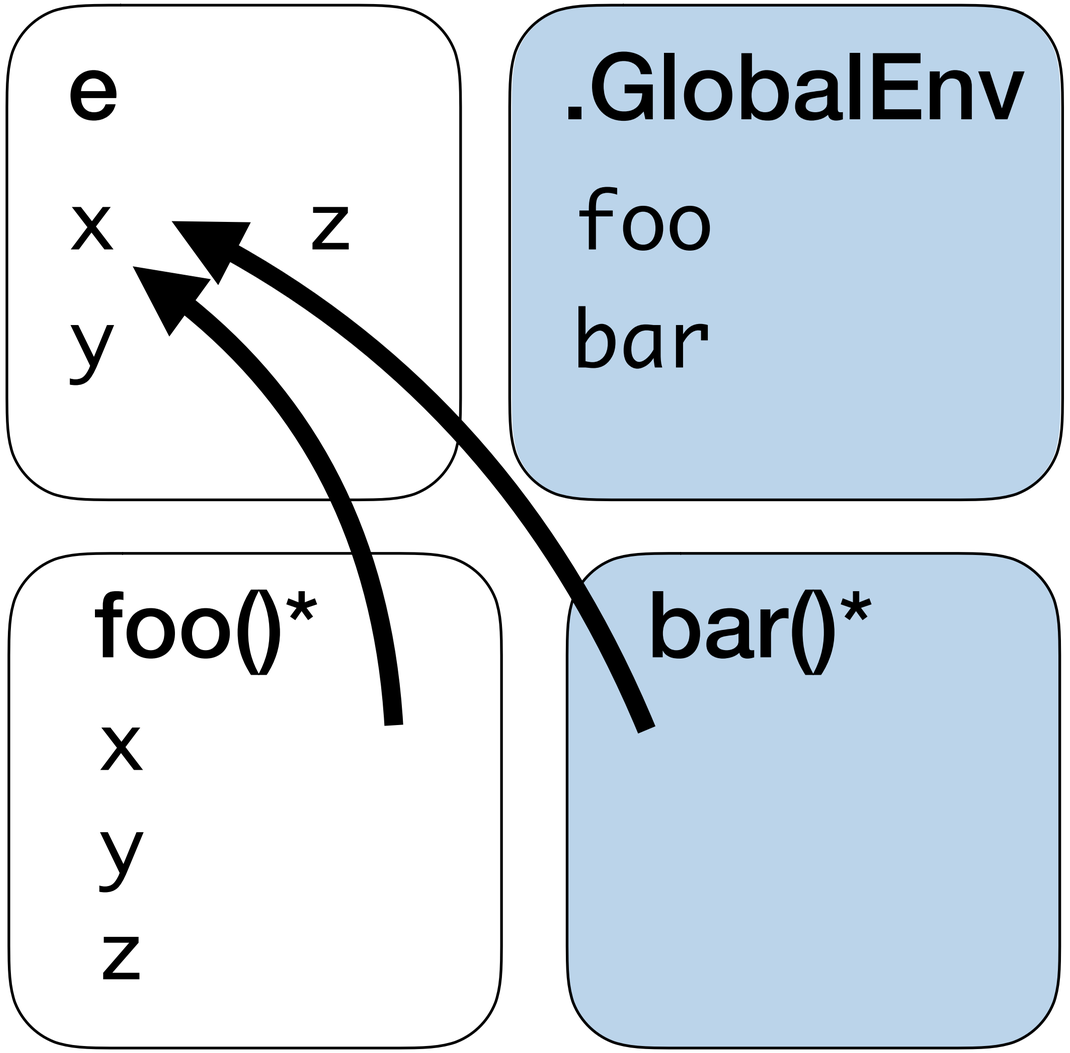
Will this work?



```
foo <- function() {
  x <- 1
  y <- 2
  z <- 3
}

bar <- function() {
  x + 1
}

foo()
bar()
```



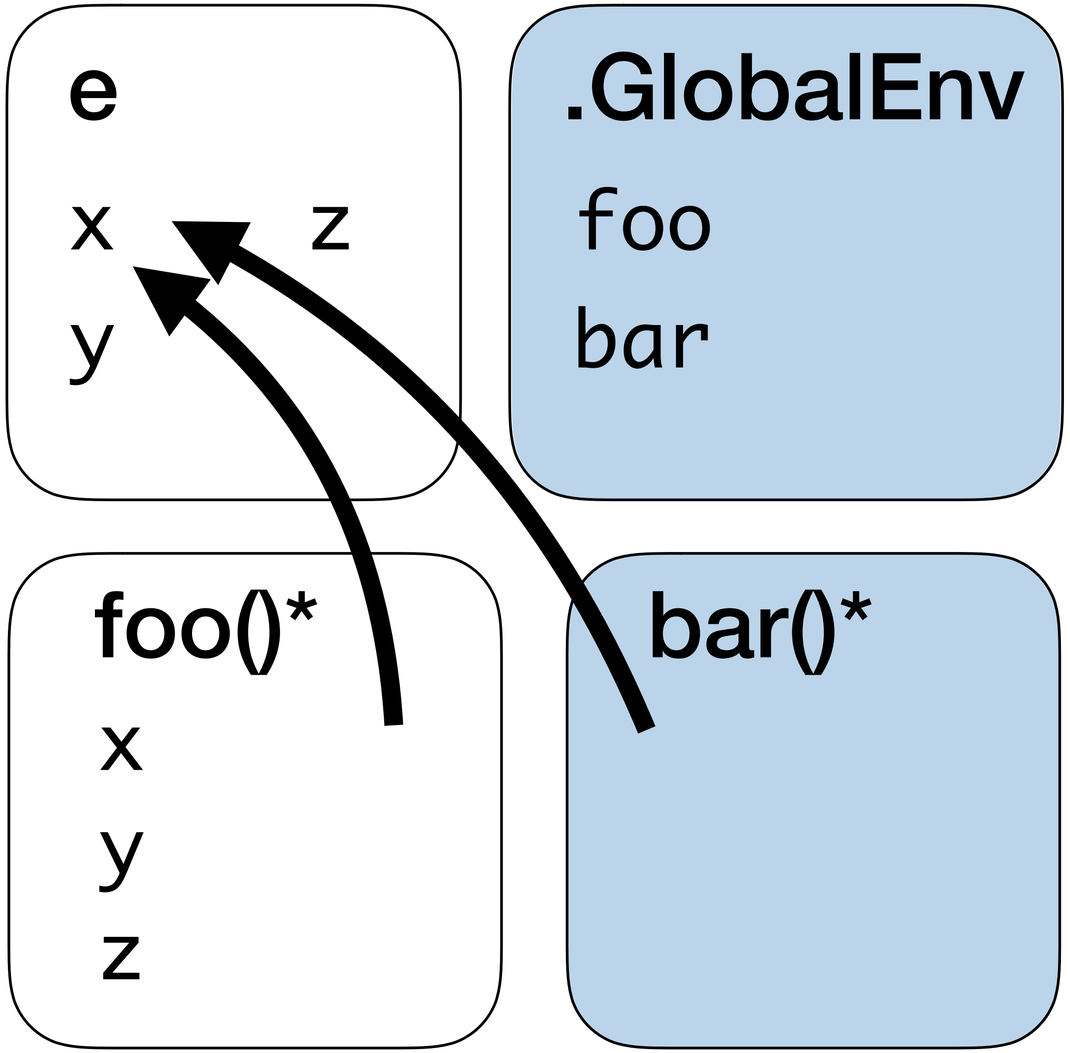
```
e <- new.env()

foo <- function() {
  e$x <- 1
  e$y <- 2
  e$z <- 3
}

bar <- function() {
  e$x + 1
}

foo()
bar()
```



1. Create new env

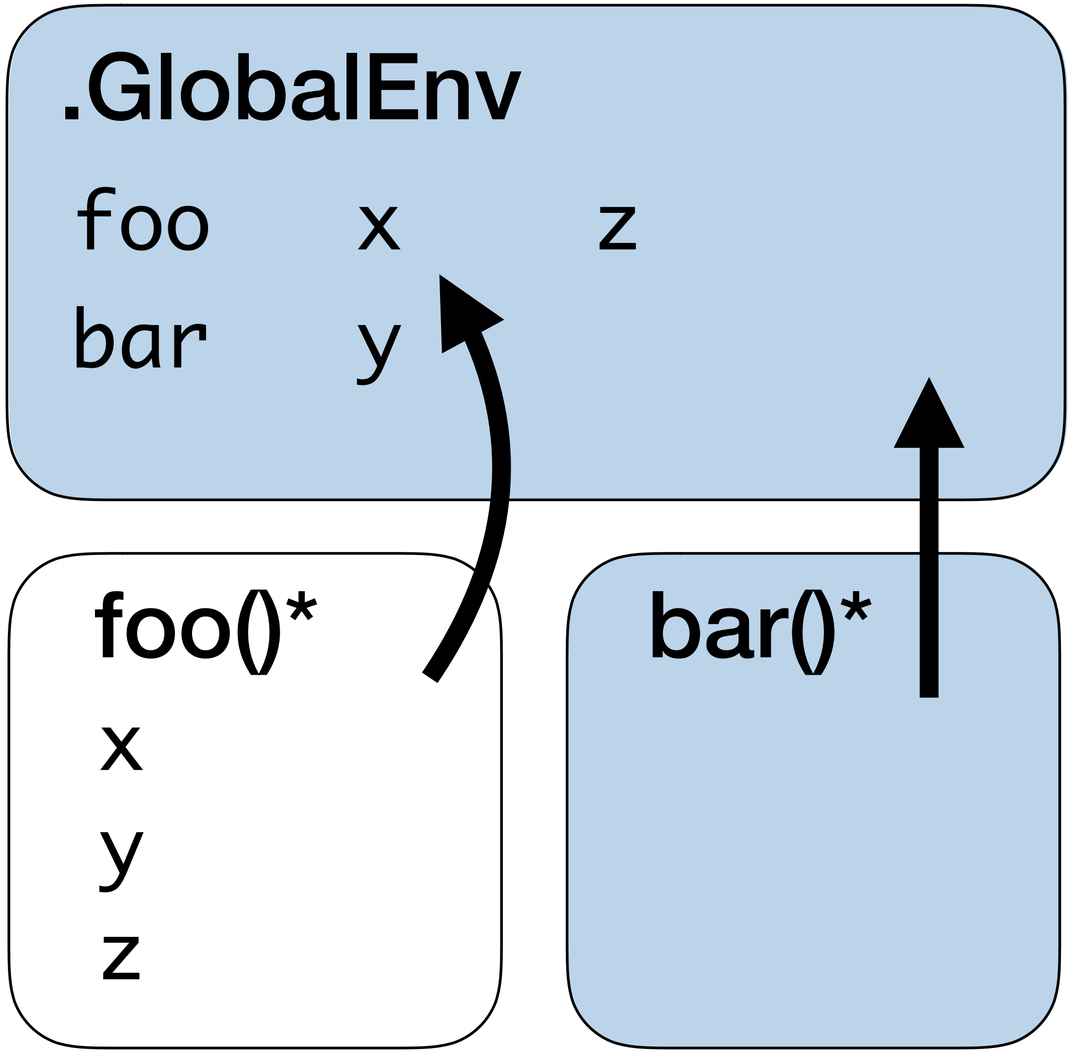
```
e <- new.env()

foo <- function() {
  e$x <- 1
  e$y <- 2
  e$z <- 3
}

bar <- function() {
  e$x + 1
}

foo()
bar()
```

NOT SELF-CONTAINED



1. Create new env

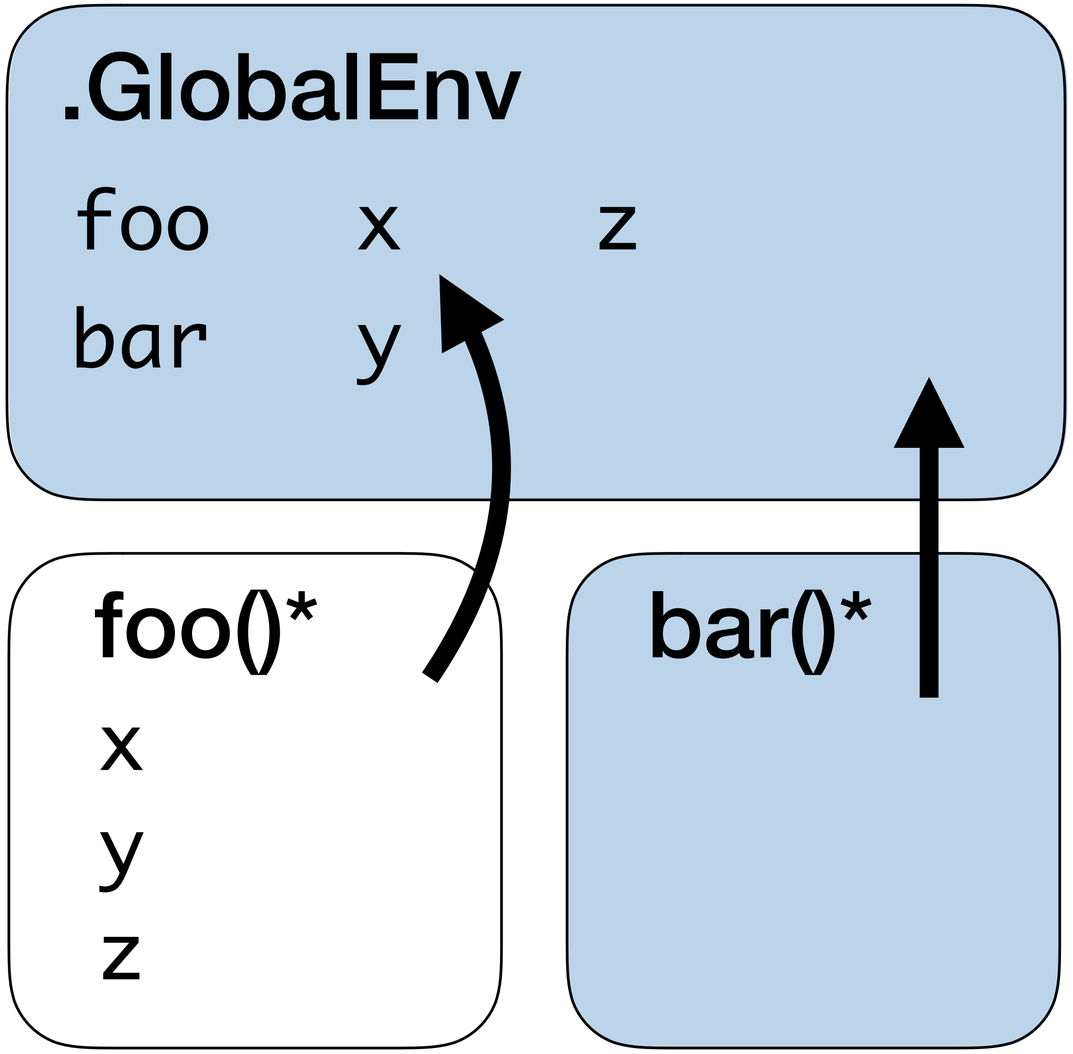
```
foo <- function() {
  assign("x", value = 1, pos = 1)
  assign("y", value = 2, pos = 1)
  assign("z", value = 3, pos = 1)
}

bar <- function() {
  x + 1
}

foo()
bar()
```



2. Assign to parent env

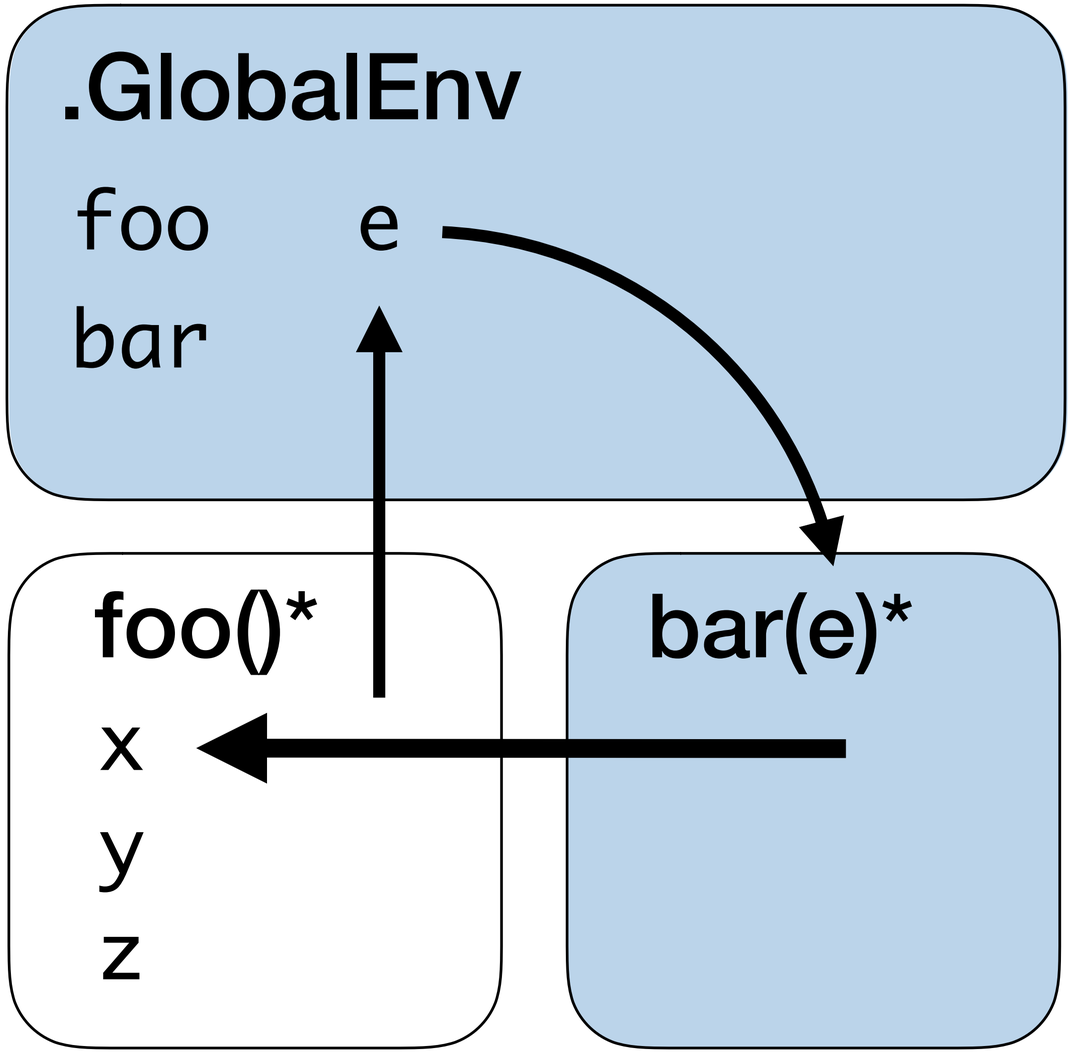
```
foo <- function() {
  assign("x", value = 1, pos = 1)
  assign("y", value = 2, pos = 1)
  assign("z", value = 3, pos = 1)
}

bar <- function() {
  x + 1
}

foo()
bar()
```

SIDE EFFECTS



2. Assign to parent env

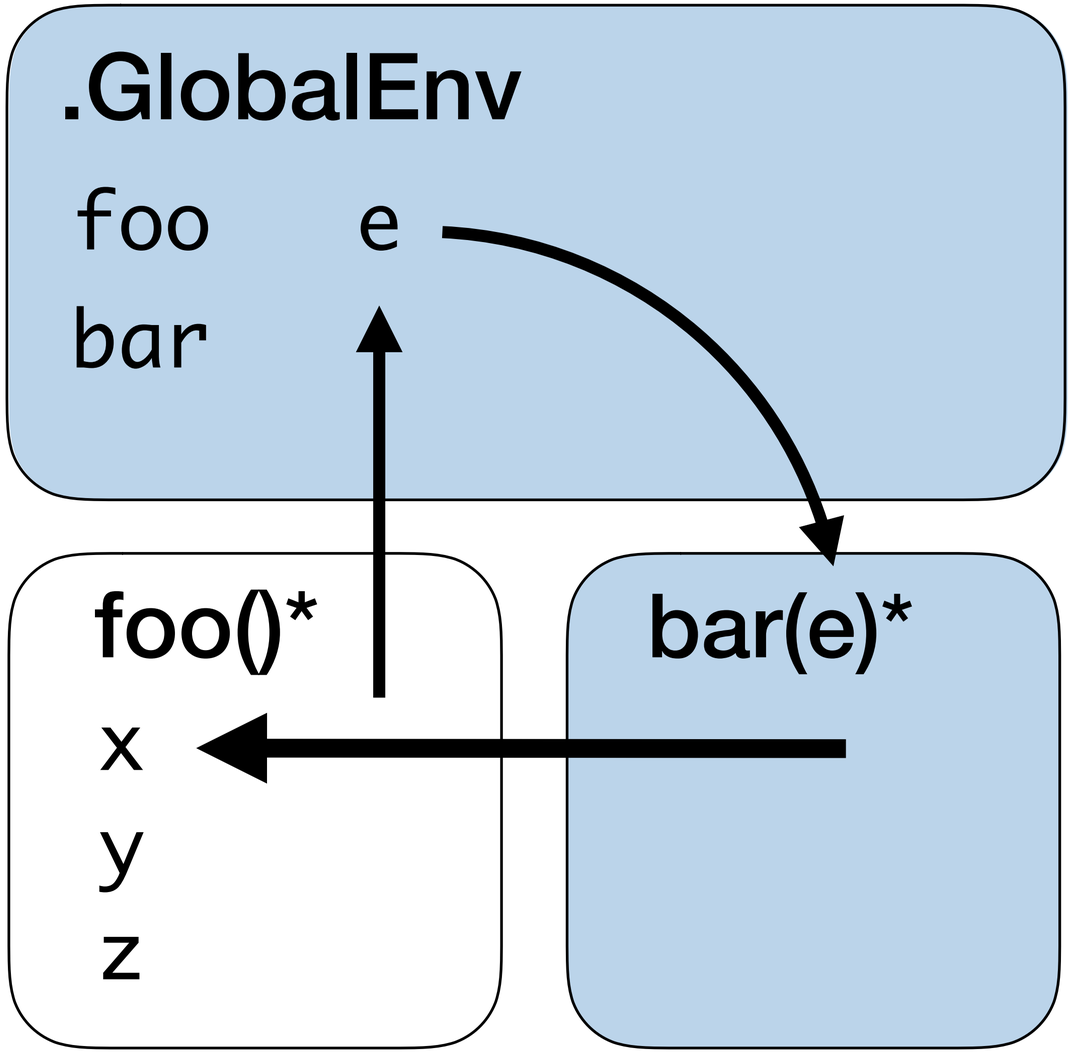
```
foo <- function() {
  x <- 1
  y <- 2
  z <- 3
  environment()
}

bar <- function(e) {
  x <- get("x", envir = e)
  x + 1
}

env <- foo()
bar(env)
```



3. Get from environment

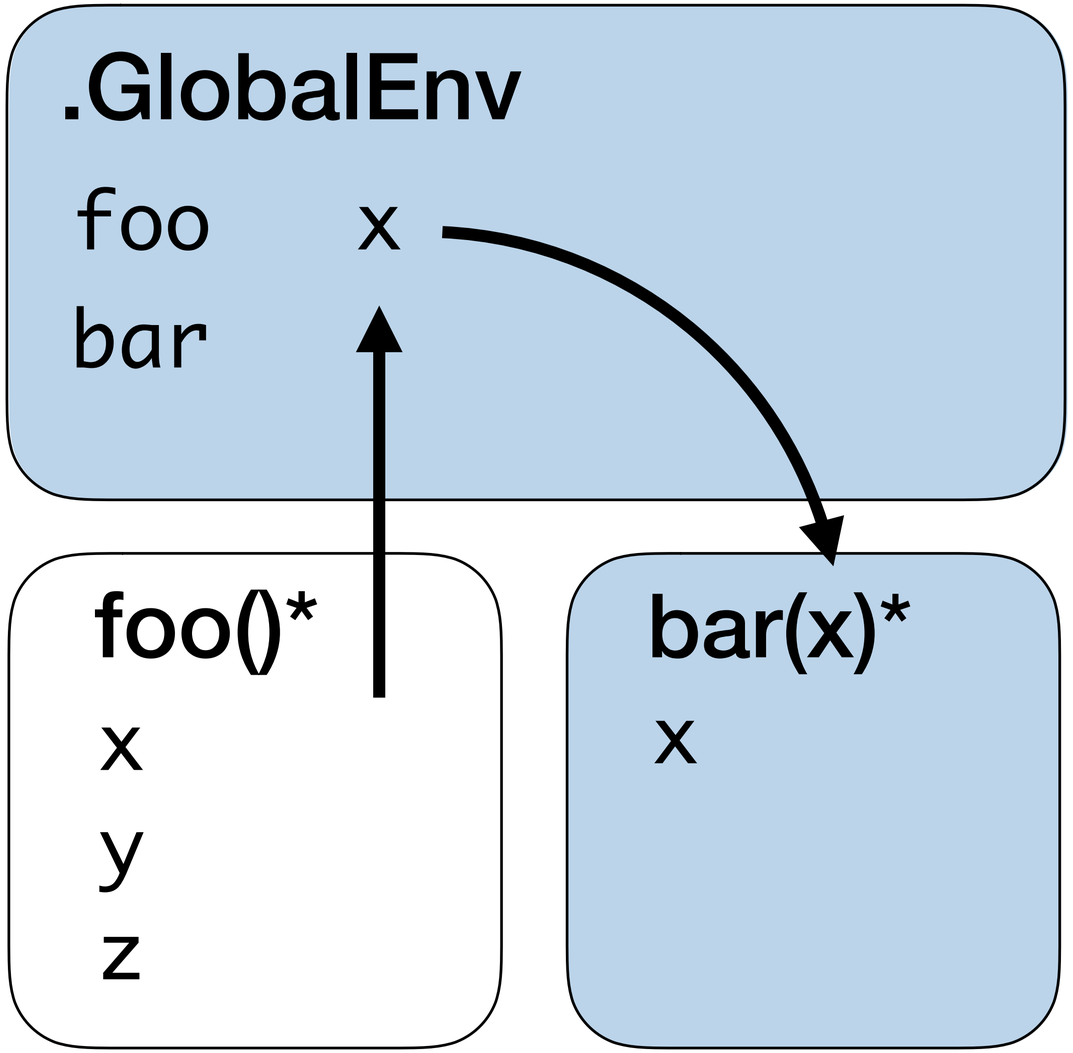
```
foo <- function() {
  x <- 1
  y <- 2
  z <- 3
  environment()
}

bar <- function(e) {
  x <- get("x", envir = e)
  x + 1
}

env <- foo()
bar(env)
```

OVERLY COMPLICATED



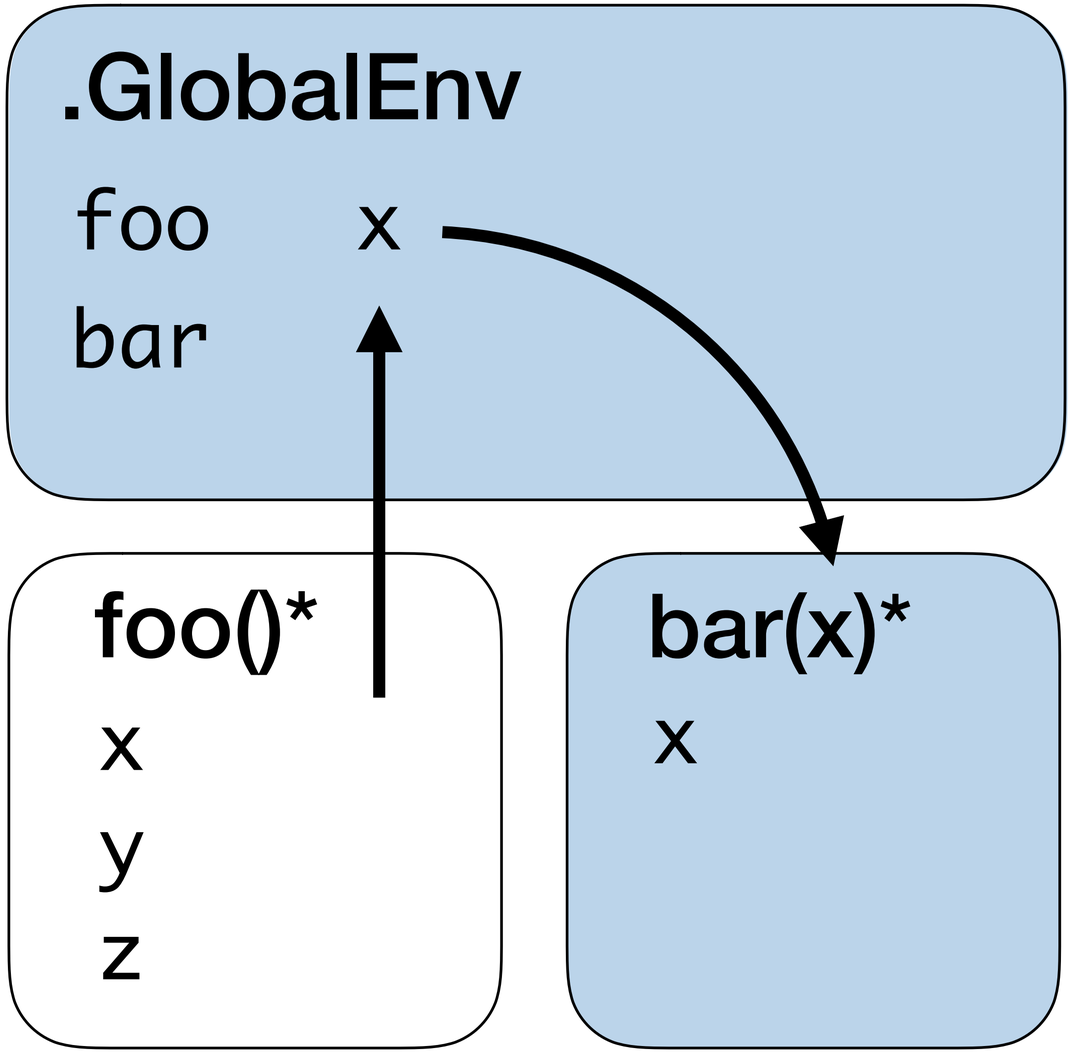
3. Get from environment

```
foo <- function() {
  x <- 1
  y <- 2
  z <- 3
  x
}

bar <- function(a) {
  a + 1
}

q <- foo()
bar(q)
```



4. Return values, arguments

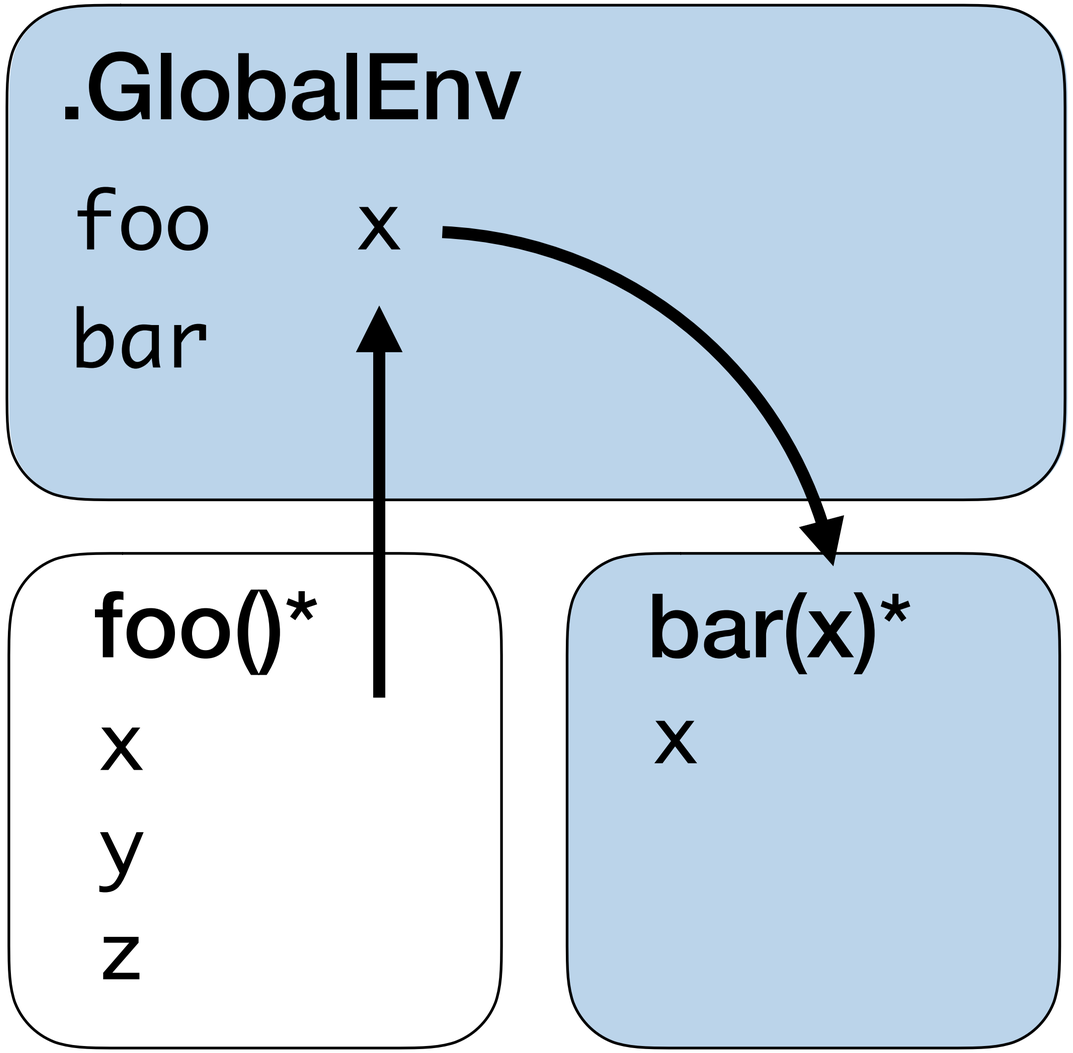
```
foo <- function() {
  x <- 1
  y <- 2
  z <- 3
  x
}

bar <- function(a) {
  a + 1
}
```




```
bar(foo())
```

4. Return values, arguments

```
foo <- function() {
  x <- 1
  y <- 2
  z <- 3
  x
}

bar <- function(a) {
  a + 1
}
```




```
foo() %>% bar()
```

4. Return values, arguments

The best **functions** should:

1. Collect input as an argument
2. Return output as a return value

The best **modules** should:

1. Collect input as an argument
2. Return output as a return value

…especially when you exchange reactive information.

Communicate with the app

Principle?

Reactive expressions are the most portable format for passing reactive information between functions

To return **reactive output** from a module

```
sliderText <- function(input, output, session){
  output$number <- renderText({input$slider})
  reactive({input$num})
}

ui <- fluidPage(
  sliderTextUI("module"),
  h2(textOutput("value")
)
server <- function(input, output) {
  num <- callModule(sliderText, "module")

  output$value <- renderText({ num() })
}
shinyApp(ui, server)
```

To return **reactive output** from a module

```
sliderText <- function(input, output, session){
  output$number <- renderText({input$slider})
  reactive({input$num}) ❶
}

ui <- fluidPage(
  sliderTextUI("module"),
  h2(textOutput("value")
)
server <- function(input, output) {
  num <- callModule(sliderText, "module")

  output$value <- renderText({ num() })
}
shinyApp(ui, server)
```

❶ Return reactive output as a reactive expression or a list of reactive expressions

To return **reactive output** from a module

```
sliderText <- function(input, output, session){
  output$number <- renderText({input$slider})
  reactive({input$num}) ❶
}

ui <- fluidPage(
  sliderTextUI("module"),
  h2(textOutput("value")
)
server <- function(input, output) {
  num <- callModule(sliderText, "module") ❷

  output$value <- renderText({ num() })
}
shinyApp(ui, server)
```

❶ Return reactive output as a reactive expression or a list of reactive expressions

❷ `callModule()` returns all of the output returned by the server function

Your Turn

Open Shiny-Modules/Exercise-3, which contains a module that loads a cvs into an app.

Modify the code in uploadModule.R so that each function returns its output in the correct way.

Then complete the app in app.R.

05:00

To pass **reactive input** to a module

```
sliderText <- function(input, output, session, show){
  output$number <- renderText({
    if (show()) input$slider
    else NULL
  })
}

ui <- fluidPage(
  checkboxInput("display", "Show Value"),
  sliderTextUI("module")
)
server <- function(input, output) {
  display <- reactive({ input$display })
  callModule(sliderText, "module", display)
}
shinyApp(ui, server)
```

To pass **reactive input** to a module

```
sliderText<-function(input,output,session,show){
  output$number <- renderText({
    if (show()) input$slider
    else NULL
  })
}
ui <- fluidPage(
  checkboxInput("display", "Show Value"),
  sliderTextUI("module")
)
server <- function(input, output) {
  display <- reactive({ input$display }) ❶
  callModule(sliderText, "module", display)
}
shinyApp(ui, server)
```

❶ Wrap the input as a reactive expression, e.g.

```
foo <- reactive({ rv$foo })
```

To pass **reactive input** to a module

```
sliderText<-function(input,output,session,show){
  output$number <- renderText({
    if (show()) input$slider
    else NULL
  })
}
ui <- fluidPage(
  checkboxInput("display", "Show Value"),
  sliderTextUI("module")
)
server <- function(input, output) {
  display <- reactive({ input$display }) ❶
  callModule(sliderText, "module", display) ❷
}
shinyApp(ui, server)
```

❶ Wrap the input as a reactive expression, e.g.

`foo <- reactive({ rv$foo })`

❷ Pass the reactive expression to the module as an argument, e.g. `module(data = foo)`.

Notice that you do not use parentheses to call the value of the reactive expression when you pass the argument, e.g. `foo()`.

To pass **reactive input** to a module

```
sliderText<-function(input,output,session,show){
  output$number <- renderText({
    if (show()) input$slider ❸
    else NULL
  })
}
ui <- fluidPage(
  checkboxInput("display", "Show Value"),
  sliderTextUI("module")
)
server <- function(input, output) {
  display <- reactive({ input$display }) ❶
  callModule(sliderText, "module", display) ❷
}
shinyApp(ui, server)
```

❶ Wrap the input as a reactive expression, e.g.

`foo <- reactive({ rv$foo })`

❷ Pass the reactive expression to the module as an argument, e.g. `module(data = foo)`.

Notice that you do not use parentheses to call the value of the reactive expression when you pass the argument, e.g. `foo()`.

❸ Treat the argument as a reactive expression within the function, e.g. use parentheses: `data()`.

Your Turn

Open Shiny-Modules/Exercise-4, which contains a module that downloads data from an app.

Modify the code in downloadModule.R so that the server function correctly collects and uses `datafile` and `input$row.names` from the parent app.

Then complete the app in app.R.

Special cases

Nested uses

Wrap inner module ids with ns() within UI function.

```
sliderTextUI <- function(id){
  ns <- NS(id)
  tagList(
    sliderInput(ns("slider"), "Slide me", 0, 100, 1),
    textOutput(ns("number")),
    innerModuleUI(ns("inner"))
  )
}
sliderText <- function(input, output, session){
  callModule(innerModule, "inner"))
  output$number <- renderText({
    input$slider
  })
}
```

Rendered UI

Access the namespace id with **session$ns** in the module server function

```
maybeButtonModuleUI <- function(id) {
  ns <- NS(id)
  tagList(
    checkboxInput(ns("check"), "Display a button?"),
    uiOutput(ns("UI"))
  )
}

maybeButtonModule <- function(input, output, session) {
  ns <- session$ns

  output$UI <- renderUI({
    if (input$check)
      actionButton(ns("button"), "Click me")
    else
      NULL
  })
}
```

Thank you

Read more at

shiny.rstudio.com/articles/modules.html