Modules

Mine Çetinkaya-Rundel

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

What is a module?

- A module is a self-contained, composable component of a Shiny app
  - self-contained like a function
  - can be combined to make an app
- Have their own UI and server (in addition to the app UI and server)
- Useful for reusability
  - rather than copy and paste code, you can use modules to help manage the pieces that will be repeated throughout a single app or across multiple apps
  - can be bundled into packages
- Essential for managing code complexity in larger apps

Limitations to just functionalizing

- It's possible to write UI-generating functions and call them from your app's UI, and you write functions for the server that define outputs and create reactive expressions
- However you must make sure your functions generate input and output IDs that don't collide since input and output IDs in Shiny apps share a global namespace, meaning, each ID must be unique across the entire app
- **Solution:** Namespaces! Modules add namespacing to Shiny UI and server logic

"Roughly, hygienic macro expansion is desirable for the same reason as lexical scope: both enable local reasoning about binding so that program fragments compose reliably."

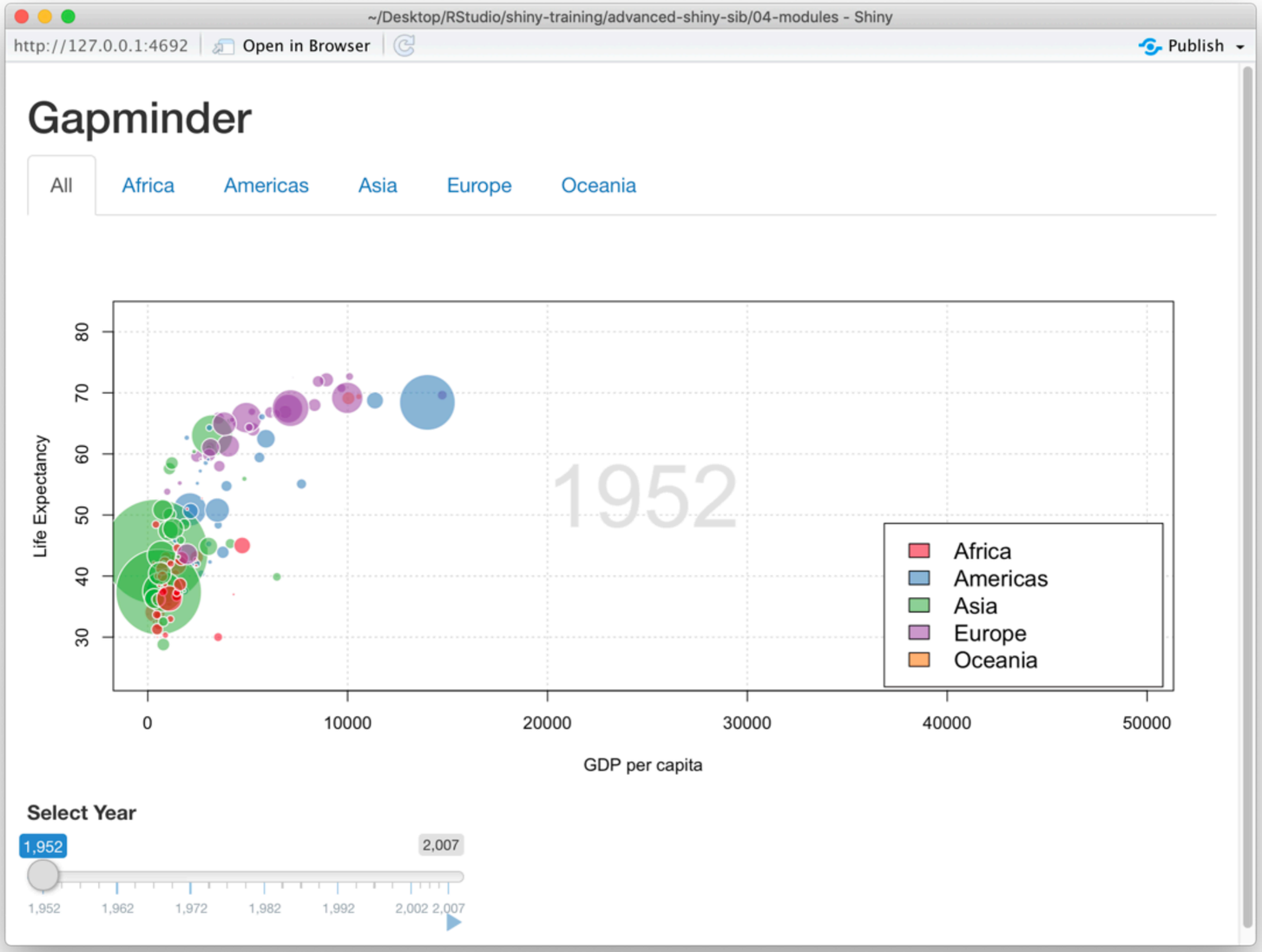
–Matthew Flatt

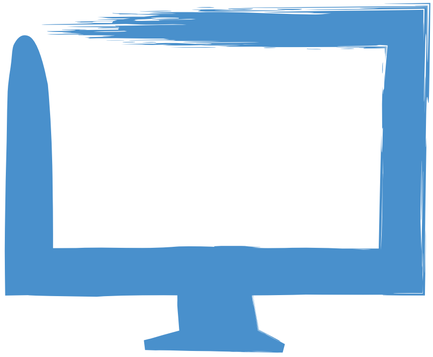
Shiny modules

"Roughly, ~~hygienic macro expansion~~ is desirable for the same reason as lexical scope: both enable **local reasoning** about binding so that program fragments **compose reliably**."

–Matthew Flatt

Demo


~/Desktop/RStudio/shiny-training/advanced-shiny-sib/04-modules - Shiny
http://127.0.0.1:4692
Open in Browser
Publish
Gapminder
All
Africa
Americas
Asia
Europe
Oceania
Life Expectancy
80
70
60
50
40
30
1952
Africa
Americas
Asia
Europe
Oceania
0
10000
20000
30000
40000
50000
GDP per capita
Select Year
1,952
2,007
1,952
1,962
1,972
1,982
1,992
2,002
2,007



Ladder of progression

- Step 1. Use modules to break large monolithic apps into manageable pieces
- Step 2. Create reusable modules
- Step 3. Nest modules

Anatomy of a Shiny module

What's in a module?

```r
library(shiny)
name_of_module_UI <- function(id, label = "Some label") {
  # Create a namespace function using the provided id
  ns <- NS(id)
  # UI elements go here
  tagList(
    ...
  )
}

name_of_module <- function(input, output, session, …) {
  # Server logic goes here
}
```

Module vs. app

- Similarities:
  - Inputs in UI can be accessed in server with `input$`
  - Outputs in UI can be defined in server with `output$`
- Differences:
  - Inputs/outputs cannot be directly accessed from outside the module namespace
  - If a module needs to use a reactive expression, take the reactive expression as a function parameter. If a module wants to return reactive expressions to the calling app, then return a list of reactive expressions from the function
  - If a module needs to access an input that isn't part of the module, the containing app should pass the input value wrapped in a reactive expression

Module UI

- A function
- Takes, as input, an id that gets pre-pended to all HTML element ids with a helper function: `NS( )`
- Can also have additional parameters

Module server

- Includes the code needed for your module
- Looks almost identical to the app server function, except that you may have additional parameters
- App server function is automatically invoked by Shiny; module server function must be invoked by the app author

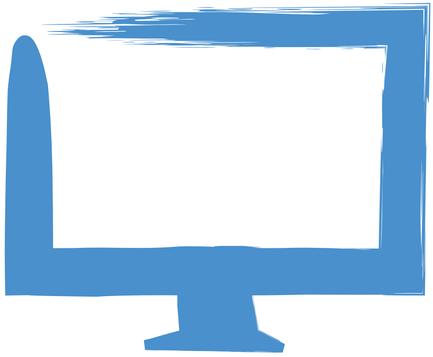
Calling the module

- In the app UI:
  - Include the module UI with `name_of_module_UI("id", …)`
  - Can also include other UI elements that are not included in the module
- In the app server:
  - Include the module server with `callModule(name_of_module, "id", …)`
  - Can also include other UI elements that are not included in the module
  - The id must match and must be unique among other inputs/outputs/modules at the same "scope" (either top-level ui/server, or within a parent Shiny module)

```r
ui <- fluidPage(
  …
  titlePanel("Gapminder"),
  tabsetPanel(id = "continent",
    tabPanel("All", gapModuleUI("all")),
    tabPanel("Africa", gapModuleUI("africa")),
    tabPanel("Americas", gapModuleUI("americas")),
    tabPanel("Asia", gapModuleUI("asia")),
    tabPanel("Europe", gapModuleUI("europe")),
    tabPanel("Oceania", gapModuleUI("oceania"))
  )
)
```

```r
server <- function(input, output) {
  callModule(gapModule, "all", all_data)
  callModule(gapModule, "africa", africa_data)
  callModule(gapModule, "americas", americas_data)
  callModule(gapModule, "asia", asia_data)
  callModule(gapModule, "europe", europe_data)
  callModule(gapModule, "oceania", oceania_data)
}
```

Demo



Your turn

- Open `04-modules/01-modules.R` and run it. The app has three tabs: one for each title type, showing a scatterplot and data table.
- The app is created by repeating the plotting and data table code chunks three times each.
- Modularize the app using `04-modules/02-modules.R` and `04-modules/02-moviesmodules.R` as a starting points.

10m 00s

Solution

Solutions to the previous exercises

```
> 04-modules/03-movies.R
> 04-modules/03-moviesmodule.R
```

Combining modules

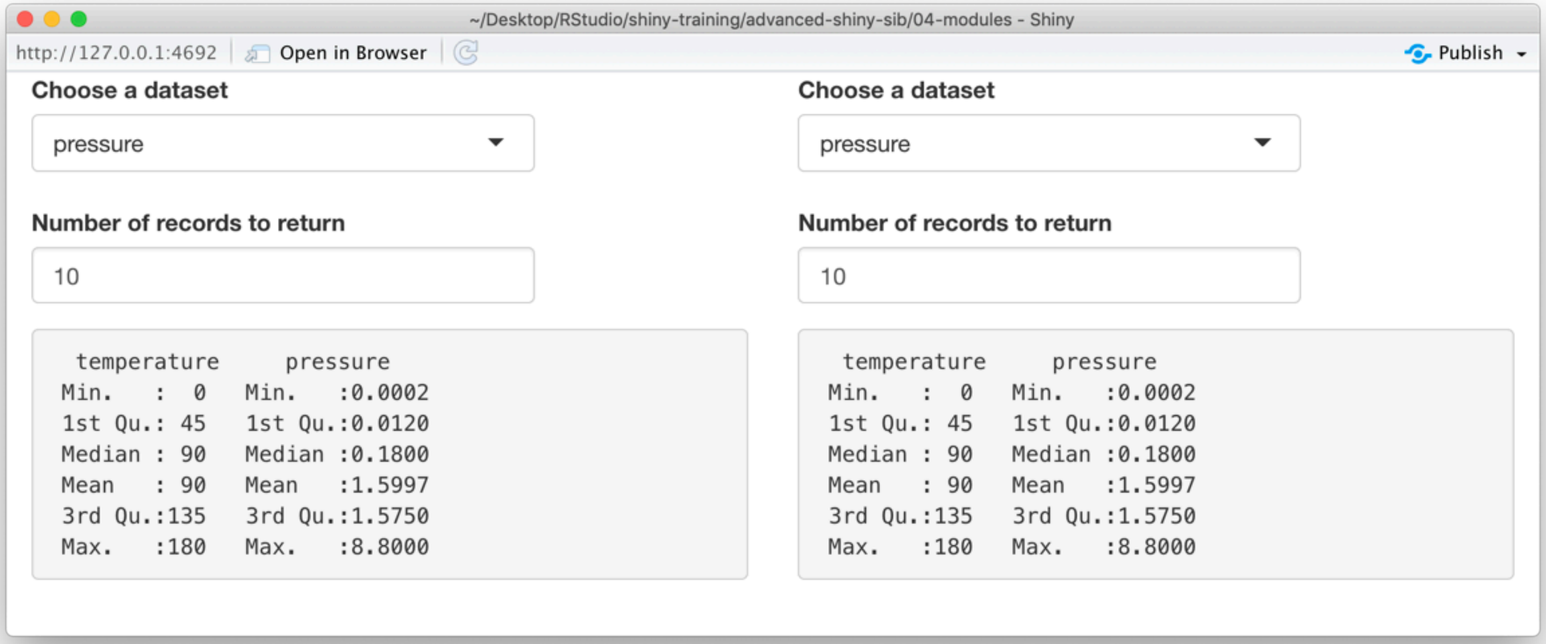
Combining modules

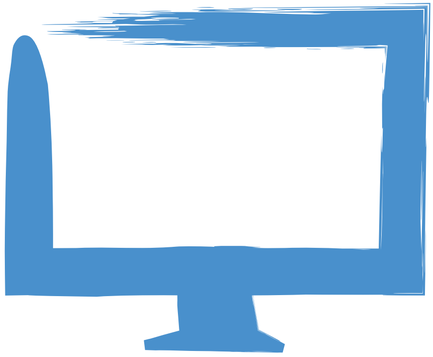
- When building an app that uses modules that depend on each other, avoid violating the sanctity of the module's namespace (similar to a function's local environment)
- If results of Module 1 will be used as inputs in Module 2, then Module 1 needs to return those results as an output, so that Module 2 does not have to “reach in and grab them”

Demo

```
> 04-modules/04-left-right.R
```

Clearly actions are repeated on the left and right for different datasets, so make use of modules.





Modules

Mine Çetinkaya-Rundel

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