Interactive data visualization

Lecture 15

Louis SIRUGUE

01/2022

Last time we saw

Shapefiles and rasters

Two main types of geolocalized datasets:

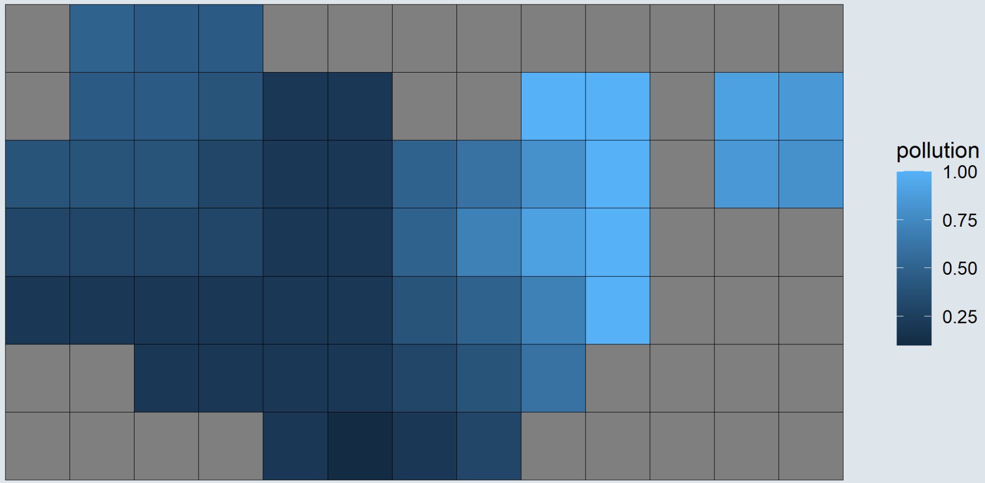
Shapefiles

- One row per entity/one column per variable
- A geometry variable with the coordinates of the points/polylines/polygons

Median income
High
Low

Rasters

- Works like a picture, with cells like pixels
- And each cell can take a given value, e.g. pollution observed from satellites

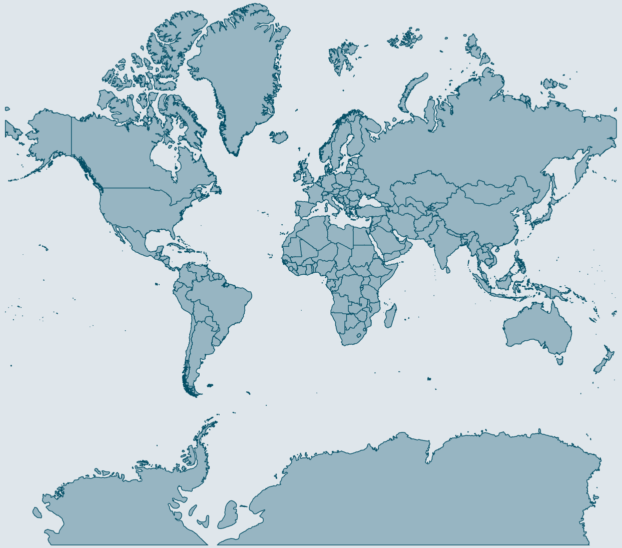


Last time we saw

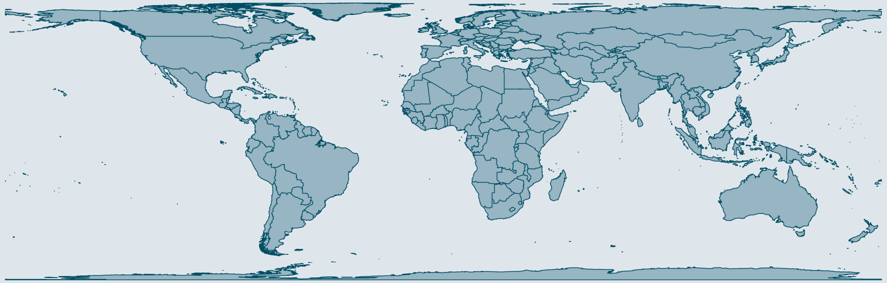
Coordinates Reference Systems

- A Coordinate Reference System (CRS) is a model of the Earth in which each location is coded using degrees
  - It allows to **project the surface of the globe on a plane**
  - But there is a **tradeoff** between preserving:

Shape (like the Mercator projection)



Scale (like the Equal-Area Cylindrical projection)



- Most projections are somewhere in between
- For France: Lambert 93 projection (EPSG:2154)

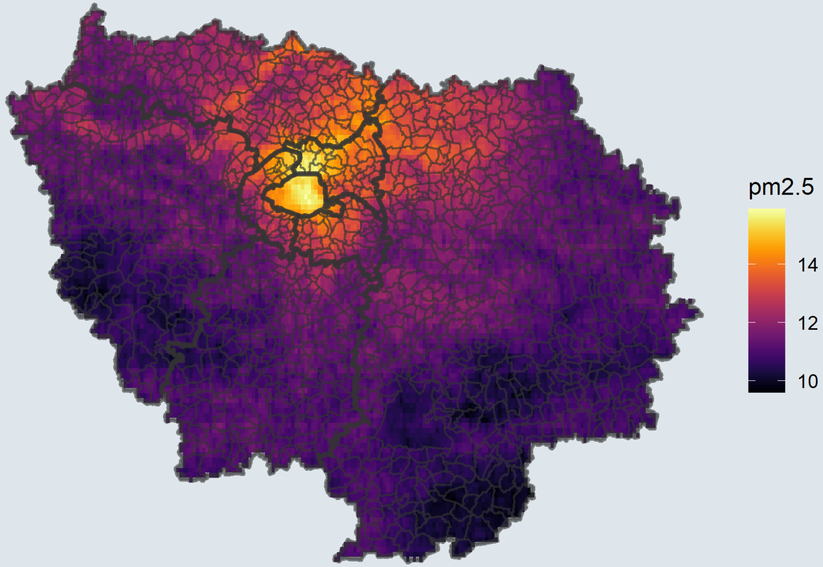
→ *First thing to do: **reprojection***

Last time we saw

Operations on geolocalized data

Zonal statistics

- Computing statistics on areas delimited by a shapefile from values of a raster
  - Project shapefile and raster the same way
  - Compute the mean/max/... of cell values



Centroids

- The centroid is the arithmetic mean position of all the points in the polygon
  - To compute distances between polygons
  - A centroid is not always within its polygon

Before we start: More on controls and interactions

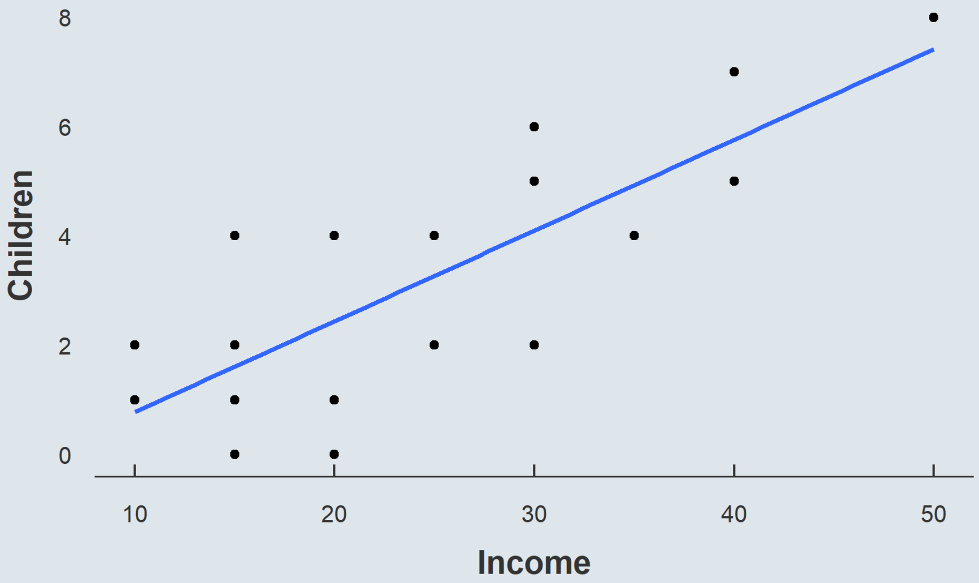
- We've seen in previous lectures that when regressing y on x:
  - **Controlling for z** allows to **net out** the relationship between x and y from how they both relate to **z**
  - **Interacting x with z** allows to **estimate how the relationship** between x and y **varies with z**

- Given what I've seen in the homeworks it seems unclear for many of you

→ So let's get back to it with some visualization

```r
library(tidyverse)
data <- read.csv("household_data.csv")
head(data, 7) # fake data
```

```
##   Income Children    Education
## 1     20        1 < Highschool
## 2     10        1 < Highschool
## 3     10        2 < Highschool
## 4     15        0 < Highschool
## 5     15        1 < Highschool
## 6     20        0 < Highschool
## 7     15        2   Highschool
```

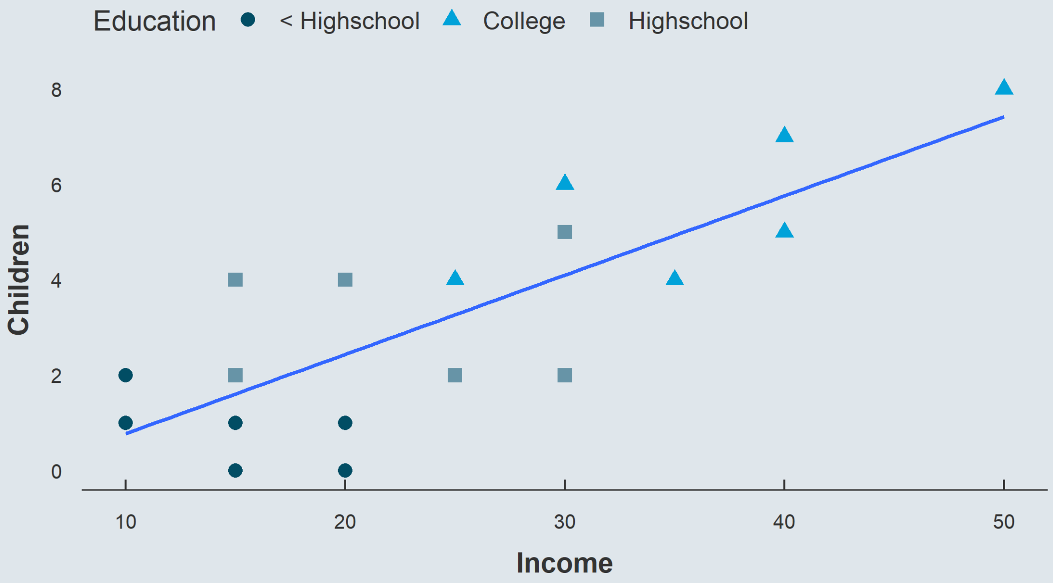


Before we start: More on controls and interactions

- There's a clear positive relationship

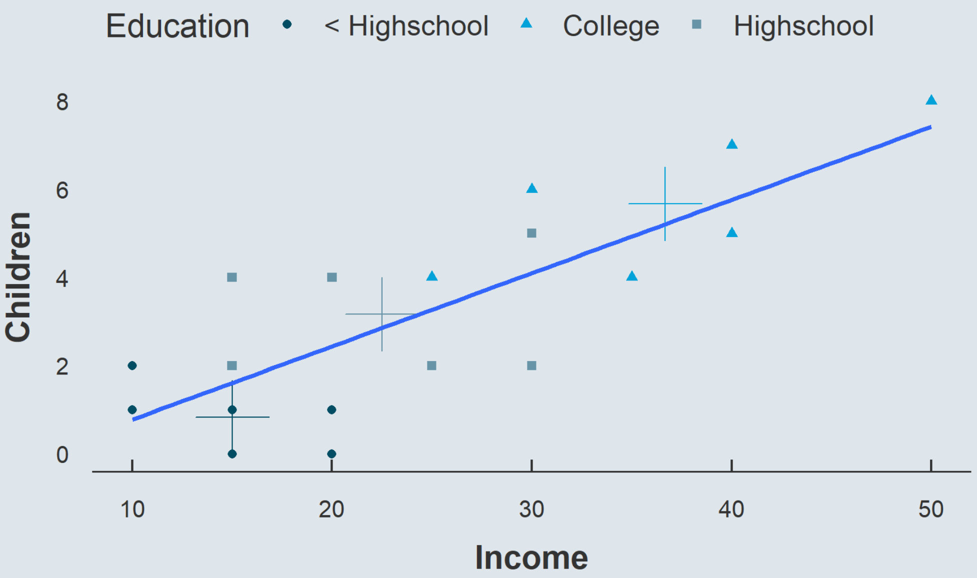
```
##               Estimate Pr(>|t|)
## (Intercept)   -0.885    0.319
## Income         0.166    0.000
```

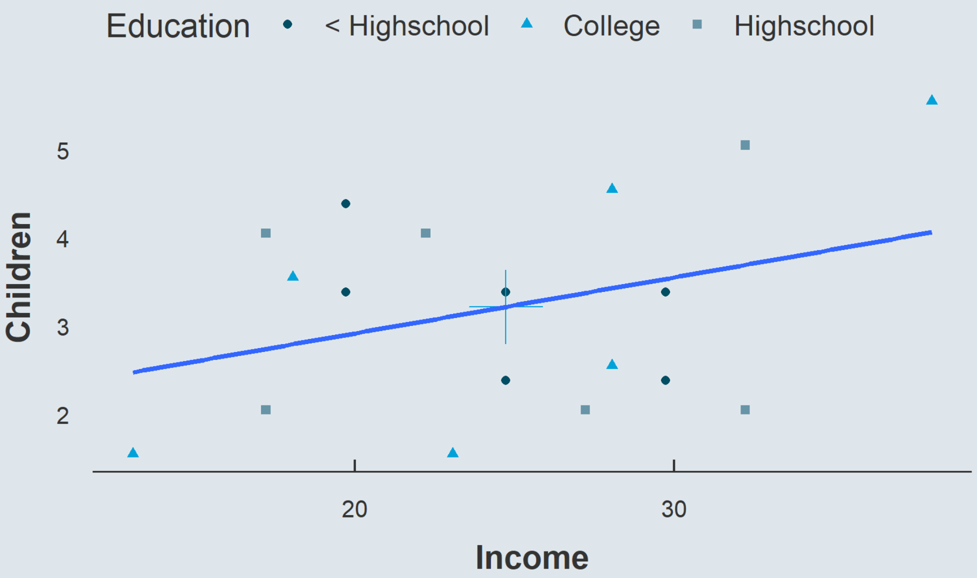
  - But what if this relationship was driven by a third variable?
  - Maybe it's just that more educated parents tend to earn more and to have more children



Before we start: More on controls and interactions

- **Controlling** for education does the same to the slope **as recentering** the graph with respect to education :





- The crosses are located at the average x and y values for each education group
  - Controlling for education shifts x and y by group such that crosses superimpose

```
##                      Estimate Pr(>|t|)
## (Intercept)            -0.120    0.892
## Income                  0.064    0.196
## EducationCollege        3.456    0.015
## EducationHighschool     1.856    0.037
```

Before we start: More on controls and interactions

- Here when we **do not control** for education:

$$Children_i = \alpha + \beta Income_i + \varepsilon_i$$

  - We estimate the overall relationship (here, significantly positive)

- But when we **control** for education:

$$Children_i = \alpha + \beta Income_i + \gamma_1 1\{Education_i = \text{Highschool}\} + \gamma_2 1\{Education_i = \text{College}\} + \varepsilon_i$$

  - We estimate the relationship net of the effect of education (here, not significant)

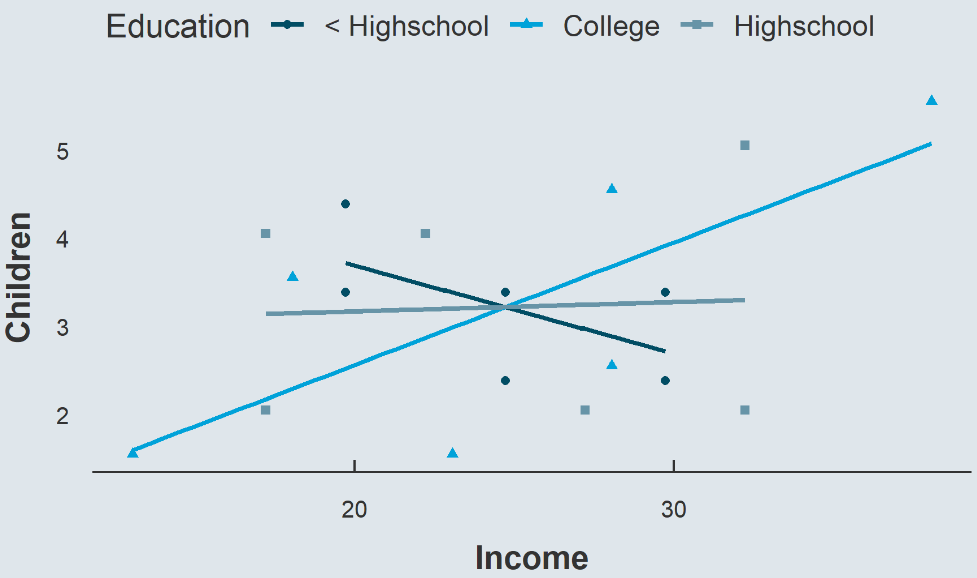
- **Interacting** the two variables is going one step further:

$$\begin{aligned} Children_i = \alpha + \beta Income_i + \gamma_1 1\{Education_i = \text{Highschool}\} + \gamma_2 1\{Education_i = \text{College}\} + \\ \delta_1 Income_i \times 1\{Education_i = \text{Highschool}\} + \delta_2 Income_i \times 1\{Education_i = \text{College}\} + \varepsilon_i \end{aligned}$$

  - It is not simply taking into account the fact that education may plays a role
  - It estimates by how much the relationship between x and y varies according to z

Before we start: More on controls and interactions

- **Interacting** income with education provides **one slope per education group**:



```
##                              Estimate Pr(>|t|)
## (Intercept)                     2.333    0.225
## Income                         -0.100    0.411
## EducationCollege               -1.768    0.553
## EducationHighschool             0.596    0.819
## Income:EducationCollege         0.239    0.095
## Income:EducationHighschool      0.111    0.445
```

- The principle is the same when the third variable is continuous:
  - Controlling nets out the slope from how the third variable enters the relationship
  - Interacting gives by how much the slope changes on expectation when the third variable increases by 1
  - And we can control for/interact with multiple third variables

Today: Interactive data visualization

1. Introduction to shiny apps

- 1.1. General structure
- 1.2. User interface
- 1.3. Server
- 1.4. Layout

2. Our first shiny app

- 2.1. Import data in Shiny
- 2.2. Interactive plot
- 2.3. Interactive regression results

3. More advanced tools

- 3.1. Input randomization
- 3.2. HTML formatting

4. Wrap up!

Today: Interactive data visualization

1. Introduction to shiny apps

- 1.1. General structure
- 1.2. User interface
- 1.3. Server
- 1.4. Layout

1. Introduction to shiny apps

1.1. General structure

- **Shiny** is an R package that makes it easy to build **interactive web apps** straight from R:
  - Shiny app to find the colleges that fit your criteria
  - Shiny app to visualize data on movies
  - The online quizzes of this course

- To make a Shiny app you should create an R script and name it **app.R**
- You shiny app should contain two components
  1. The **user interface:** What is displayed on the screen, what the user can interact with
  2. The **server:** Where the calculations are made to display the interactive components accordingly

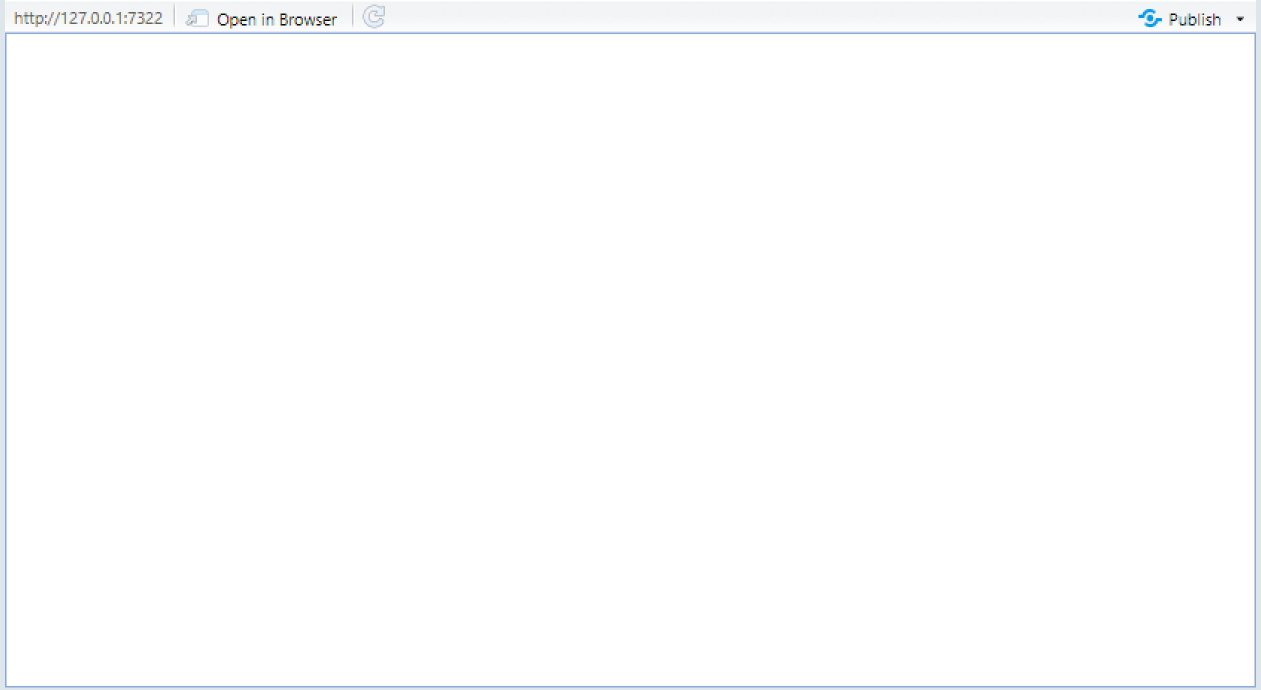
```r
library(shiny)

ui <- fluidPage()
server <- function(input, output) {}

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

→ R will detect that you are creating a shiny app and you will have access to the ▶ Run App ▾ button

1. Introduction to shiny apps

1.1. General structure

- You can already click on it to view you blank app:



➜ We have to program what we want to appear in the user interface

Introduction to shiny apps

1.2. User interface

- There are many different types of **input widgets** to place in the UI:
  - **numericInput():** Write a numeric input
  - **textInput():** Write a character input
  - **checkbox[Group]Input():** Box[es] to tick
  - **radioButtons():** One item to tick
  - **selectInput():** Select an item from a dropdown list
  - ...

- These functions take the following **arguments**:
  - **inputId:** The identifier of the input for use in the server function
  - **label:** The title of the input widget that will appear in the UI
  - **choices:** The list of input options for multiple choices inputs
  - **selected:** Which option is selected by default when multiple choices
  - **value:** What is filled by default in the text/numeric input boxes

→ Let's try out a few of them

1. Introduction to shiny apps

1.2. User interface

```
numericInput(inputId = "number",
             label = "Write a number",
             value = 0)
```

Write a number 0

```
textInput(inputId = "text",
          label = "Write text",
          value = "...")
```

Write text ...

```
checkboxInput(inputId = "box",
              label = "Tick the box",
              value = F)
```

□ Tick the box

```
checkboxGroupInput(inputId = "boxes",
                   label = "Boxes to check",
                   choices = c("A", "B", "C"),
                   selected = "B")
```

Boxes to check

□ A

☑ B

□ C

```
selectInput(inputId = "select",
            label = "Select option",
            choices = c("A", "B", "C"),
            selected = "A")
```

Select option

A ▾

1. Introduction to shiny apps

1.2. User interface

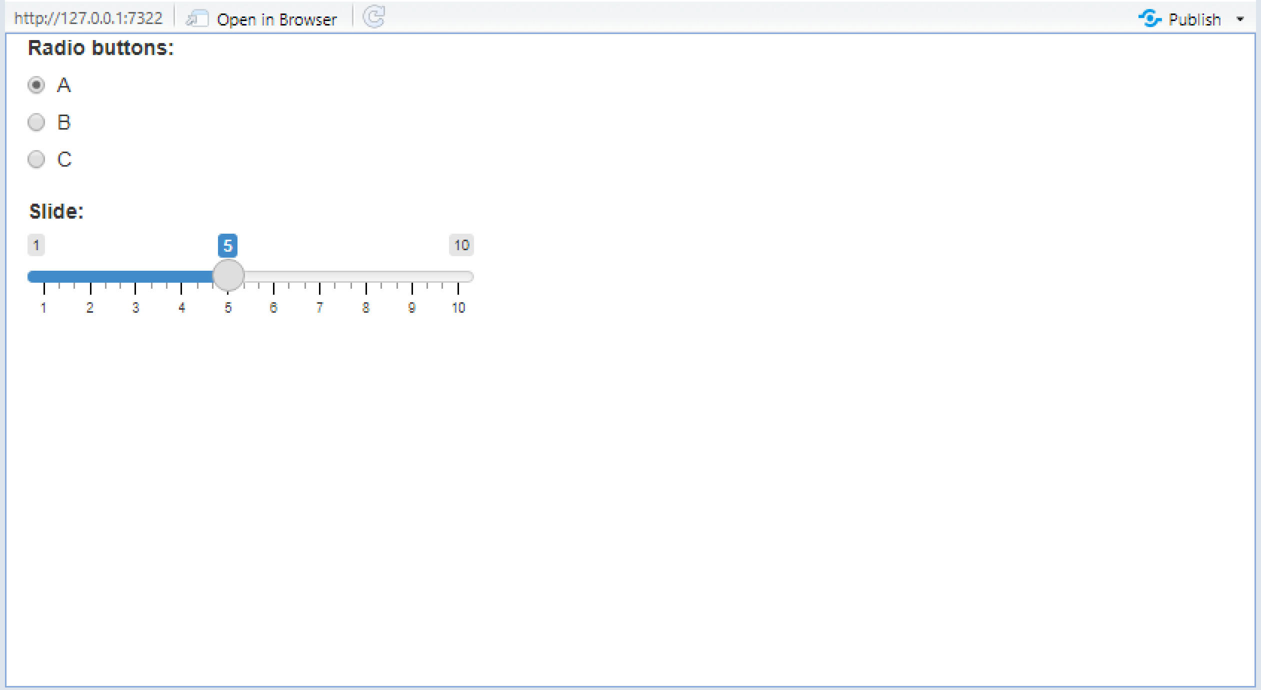
- In the UI these elements should be separated with a commas
    - Let's do a user interface with radio buttons and a slider:

```r
library(shiny)

ui <- fluidPage(
  radioButtons(inputId = "radio", label = "Radio buttons:",
               choices = c("A", "B", "C"), selected = "A"),

  sliderInput(inputId = "slider", label = "Slide:",
              min = 1, max = 10, step = 1, value = 5)
)

server <- function(input, output) {

}

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

1. Introduction to shiny apps

1.2. User interface

- Now we have some inputs but nothing happens when we use them



Radio buttons:

● A
○ B
○ C

Slide:

➜ We should make an output that will react to these inputs

1. Introduction to shiny apps

1.3. Server

- We can add a reactive table in the server function:
    - Put a standard tibble() in a **reactive({})** environment
    - Fill this table with the inputs that should be called by their id as **input$inputId**
    - Assign it to an output through **renderTable({})**

```
server <- function(input, output) {

  reactive_tibble <- reactive({
    tibble(letter = input$radio,
           number = input$slider)
  })

  output$table <- renderTable({reactive_tibble()})

}
```

- Every time an input will change, **reactive({})** function will notice it
    - And the updated table will be stored into the output named "table" through **renderTable({})**

1. Introduction to shiny apps

1.3. Server

- But for this reactive table to appear on the app, we should put it in the UI
    - Use **tableOutput("output_label")** to render a reactive table
    - Don't forget the comma!

```r
ui <- fluidPage(
  radioButtons(inputId = "radio", label = "Radio buttons:",
               choices = c("A", "B", "C"), selected = "A"),

  sliderInput(inputId = "slider", label = "Slide:",
              min = 1, max = 10, step = 1, value = 5),

  tableOutput("table")
)
```

- We now have all the components of a shiny app:
    - Some input widgets in the UI
    - Reactive functions in the server
    - The processed output in the UI

1. Introduction to shiny apps

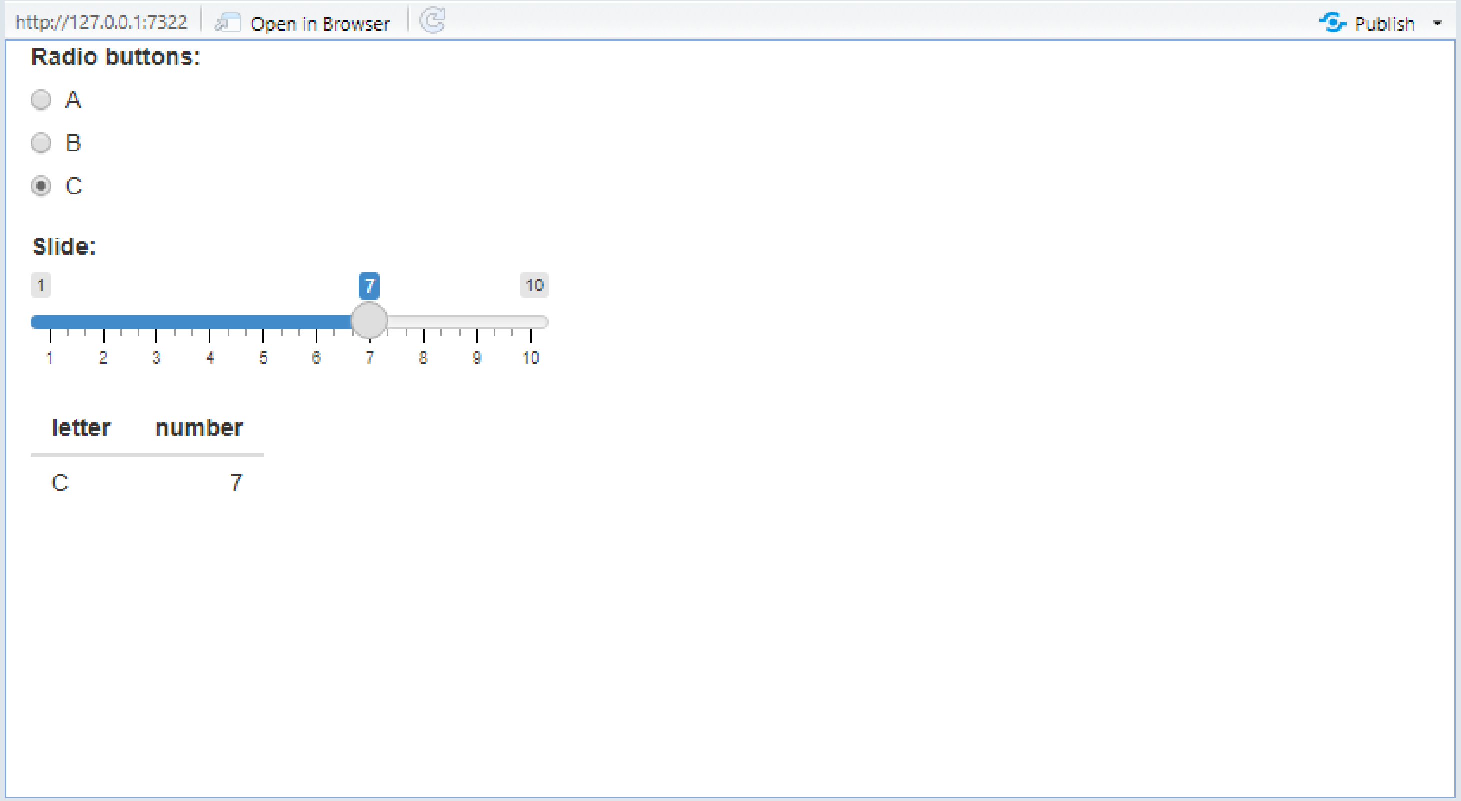
1.3. Server

```r
library(shiny)
library(tidyverse)

ui <- fluidPage(
  radioButtons(inputId = "radio", label = "Radio buttons:",
               choices = c("A", "B", "C"), selected = "A"),
  sliderInput(inputId = "slider", label = "Slide:",
              min = 1, max = 10, step = 1, value = 5),
  tableOutput("table")
)

server <- function(input, output) {
  reactive_tibble <- reactive({tibble(letter = input$radio,
                                      number = input$slider)})

  output$table <- renderTable({reactive_tibble()})
}

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

1. Introduction to shiny apps

1.3. Server

- We created an interactive table:



Radio buttons:

○ A

○ B

● C

Slide:

7

letter	number
C	7

1. Introduction to shiny apps

1.3. Server

- As you might have guessed, tableOutput() and renderTable() only work for tables
  - There are specific UI and server function for each type of interactive element

Main interactive elements in Shiny:

Interactive element	Server render function	UI output function
Table	renderTable()	tableOutput()
Plot	renderPlot()	plotOutput()
Console output	renderPrint()	verbatimTextOutput()
Text	renderText()	textOutput()
UI element	renderUI()	uiOutput()

1. Introduction to shiny apps

1.4. Layout

- Right now the presentation is not very convenient
  - Everything is stacked at the left of the page

- The sidebarLayout() allows to display:
  - A control panel on the left with the inputs
  - A main panel on the right with the reactive outputs

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(width = 3,
      #INPUTS_HERE
    ),
    mainPanel(width = 9,
      #OUTPUTS_HERE
    )
  )
)
```

1. Introduction to shiny apps

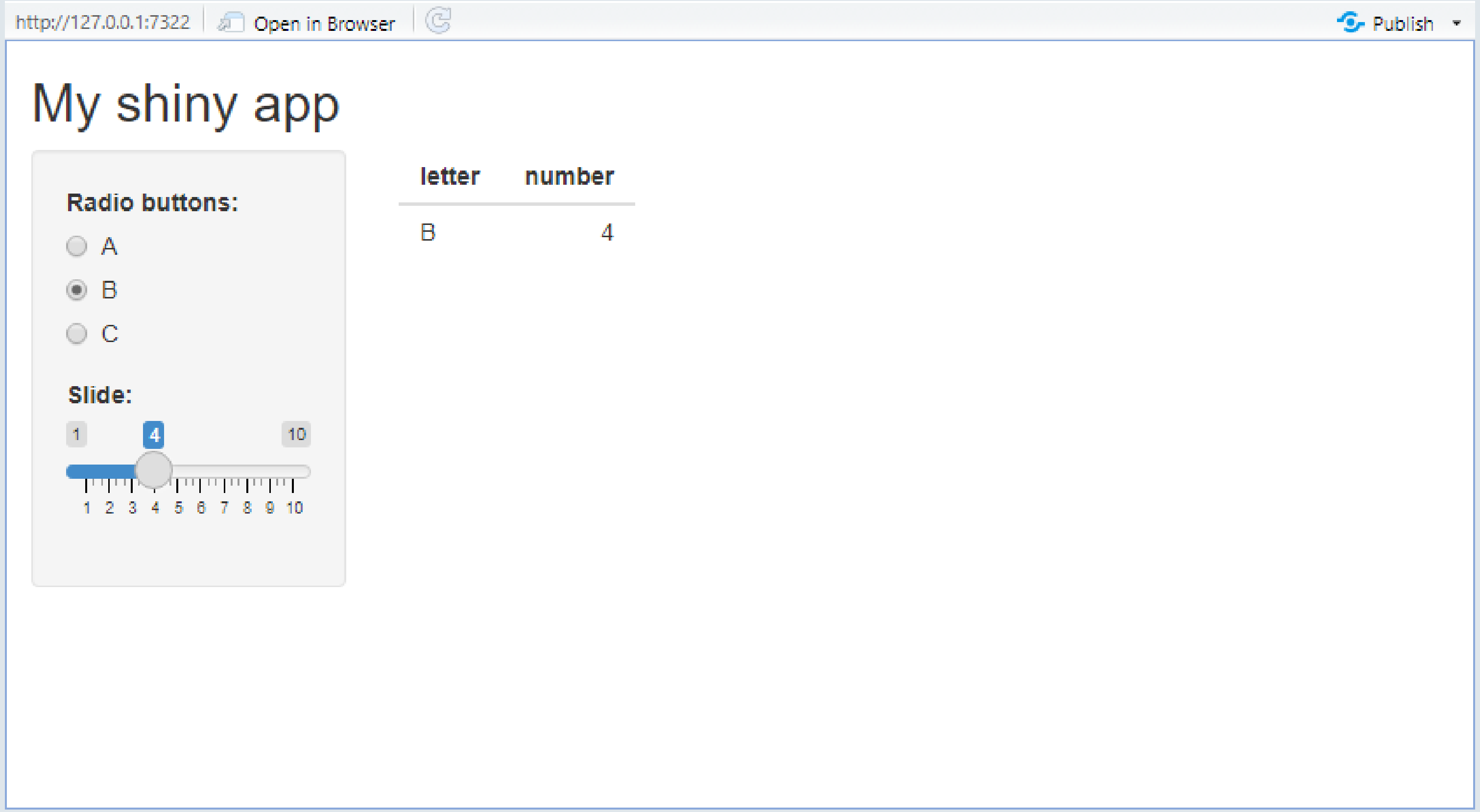
1.4. Layout

- We can also add a title to our app using the titlePanel() function:

```r
ui <- fluidPage(

  titlePanel("My shiny app"),

  sidebarLayout(

    sidebarPanel(width = 3,
      radioButtons(inputId = "radio", label = "Radio buttons:",
                   choices = c("A", "B", "C"), selected = "A"),
      sliderInput(inputId = "slider", label = "Slide:",
                  min = 1, max = 10, step = 1, value = 5)
    ),

    mainPanel(width = 9, tableOutput("table"))
  )

)
```

1. Introduction to shiny apps

1.4. Layout



My shiny app

Radio buttons:

○ A
● B
○ C

Slide:

4

letter	number
B	4

Overview

1. Introduction to shiny apps ✓

- 1.1. General structure
- 1.2. User interface
- 1.3. Server
- 1.4. Layout

2. Our first shiny app

- 2.1. Import data in Shiny
- 2.2. Interactive plot
- 2.3. Interactive regression results

3. More advanced tools

- 3.1. Input randomization
- 3.2. HTML formatting

4. Wrap up!

Overview

1. Introduction to shiny apps ✓

- 1.1. General structure
- 1.2. User interface
- 1.3. Server
- 1.4. Layout

2. Our first shiny app

- 2.1. Import data in Shiny
- 2.2. Interactive plot
- 2.3. Interactive regression results

2. Our first shiny app

2.1. Import data on shiny

- We now know everything we need to build our first app
  - Let's make an app that allows to visualize the relationship between department-level characteristics
  - dep_data.csv contains department characteristics at the department-year level from 2012 to 2017

```
dep_data <- as_tibble(read.csv("dep_data.csv"))
head(dep_data)
```

```
## # A tibble: 6 x 11
##   Department  Year Metropole Main.river Unemployment.rate Median.income
##   <chr>      <int>     <int>      <int>             <dbl>         <dbl>
## 1 Ain         2012         0          1              6.85        21122
## 2 Ain         2013         0          1              7.22        21490.
## 3 Ain         2014         0          1              7.12        21700.
## 4 Ain         2015         0          1              7.38        22020.
## 5 Ain         2016         0          1              7.35        22272
## 6 Ain         2017         0          1              6.78        22640
## # ... with 5 more variables: Share.single.parents <dbl>,
## #   PM2.5.concentration <dbl>, Population <int>, Log.population <dbl>,
## #   Log.median.income <dbl>
```

2. Our first shiny app

2.1. Import data on shiny

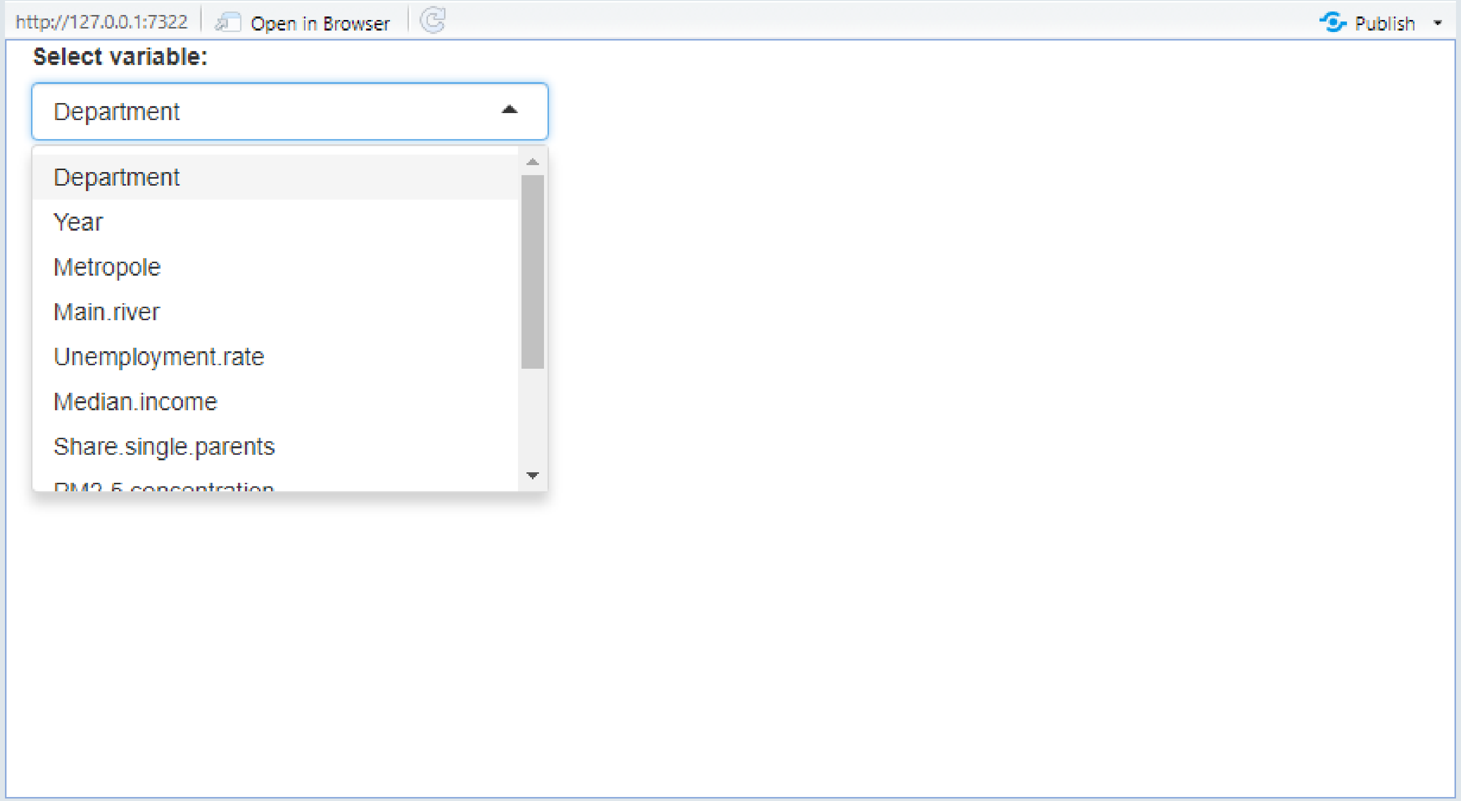
- Importing data in a Shiny app is no different than usual
  - We can simply assign it to an object before specifying the UI and the server functions
  - Every object stored at the beginning of the script is accessible by the app

```
library(shiny)
library(tidyverse)

dep_data <- as_tibble(read.csv("dep_data.csv"))

ui <- fluidPage(
  selectInput(inputId = "variable", label = "Select variable:",
              choices = names(dep_data), selected = names(dep_data)[1])
)

server <- function(input, output) {

}

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

2. Our first shiny app

2.1. Import data on shiny

- We can now access the variable of the dataset from the dropdown list:



Select variable:

Department

2. Our first shiny app

2.1. Import data on shiny

- Let's make a control panel containing:
  - Two dropdown lists for the x and y variables
  - A slider for the year of observation

- But using sliderInput() would display years with a comma, e.g., 2,012
  - To display years conveniently we should use sliderTextInput() from shinyWidgets

```
library(shiny)
library(tidyverse)
library(shinyWidgets)
```

- And we do not want the `Department` and `Year` variables to appear in our lists

```
dep_data <- as_tibble(read.csv("dep_data.csv"))
depvars <- names(dep_data)[!names(dep_data) %in% c("Department", "Year")]
```

2. Our first shiny app

2.1. Import data on shiny

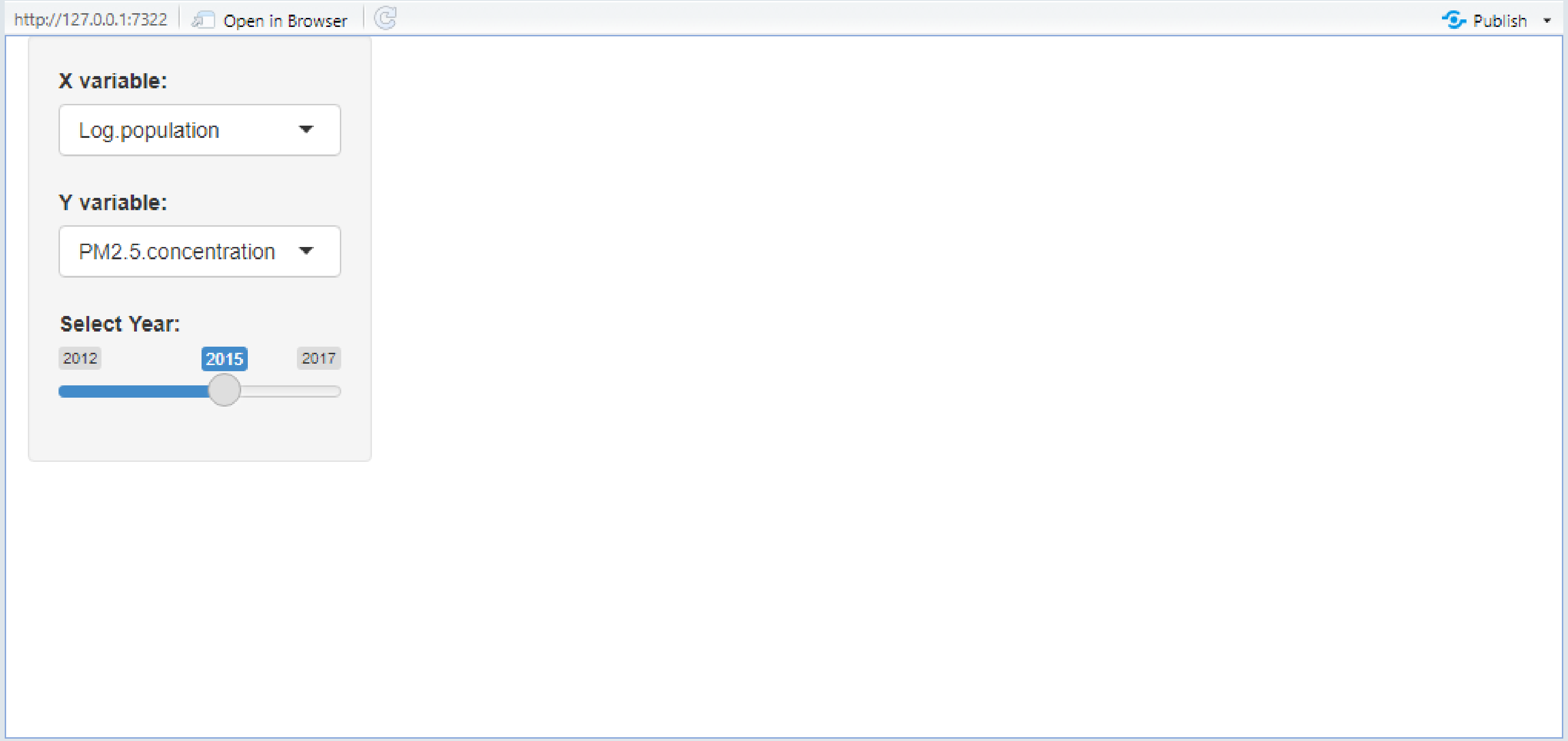
- The desired UI writes as follows

```r
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(width = 3,
      selectInput(inputId = "xvar", label = "X variable:",
                  choices = depvars, selected = "Log.population"),

      selectInput(inputId = "yvar", label = "Y variable:",
                  choices = depvars, selected = "PM2.5.concentration"),

      sliderTextInput(inputId = "year", label = "Select Year:",
                      choices = 2012:2017, selected = 2015)
    ),
    mainPanel(width = 9)
  )
)
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```

2. Our first shiny app

2.1. Import data on shiny

- We now have a control panel that is linked to our data:



X variable: Log.population

Y variable: PM2.5.concentration

Select Year: 2015

2. Our first shiny app

2.2. Interactive plot

- To do an interactive plot that reacts to these inputs we should:
    1. Program the plot in a reactive({}) environment for the inputs to be updated
    2. Put the reactive plot in a render function to render the updated plot
    3. Include the resulting output in the user interface

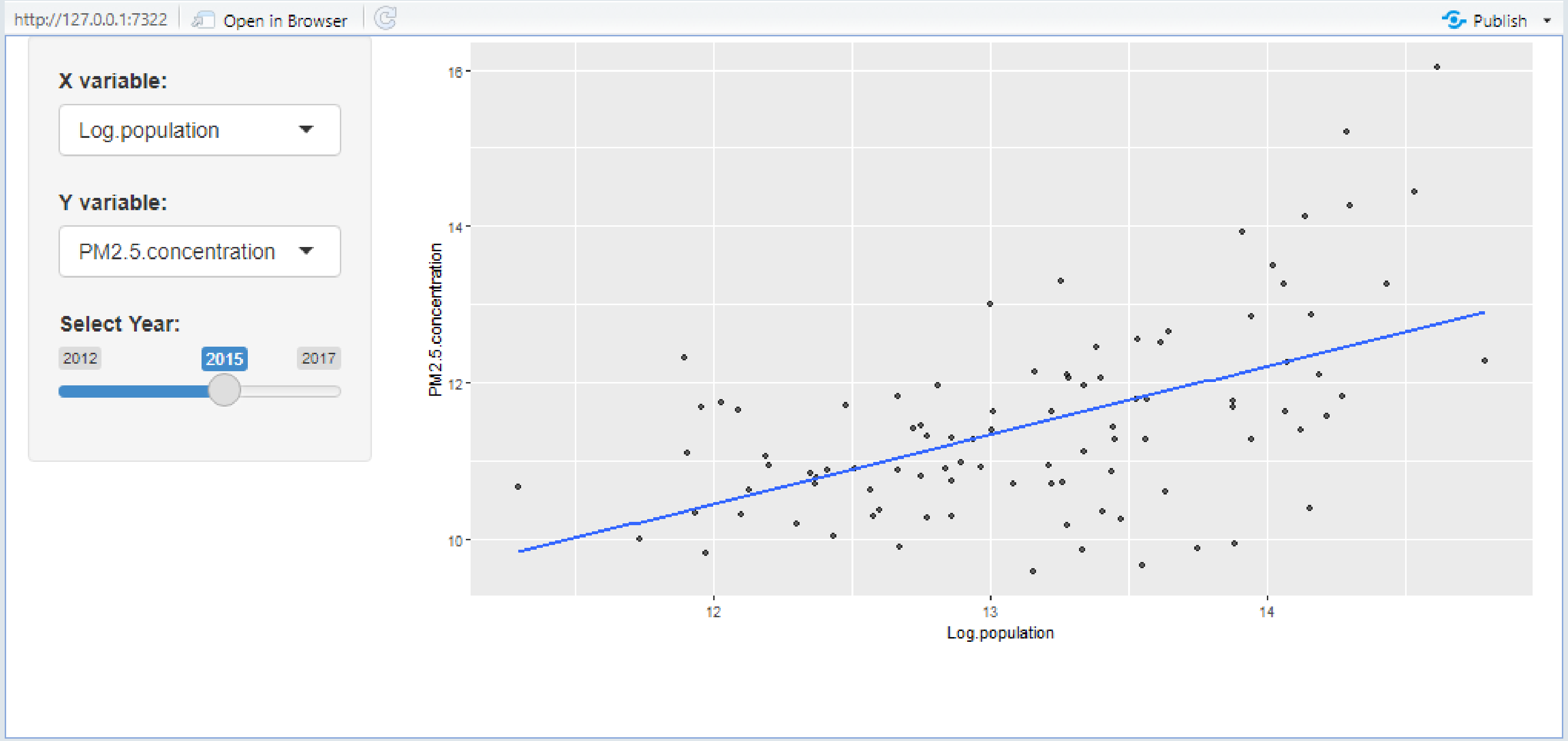
- Because in aes() `input$xvar` and `input$yvar` should be treated as variable names instead of strings, they should be put in the `get()` function

```r
server <- function(input, output) {

  reactive_plot <- reactive({
    ggplot(dep_data %>% filter(Year == input$year),
           aes(x = get(input$xvar), y = get(input$yvar))) +
      geom_point(alpha = .6) + geom_smooth(method = "lm", se = F) +
      xlab(input$xvar) + ylab(input$yvar)
  })
   output$plot <- renderPlot({reactive_plot()})
}
```

2. Our first shiny app

2.2. Interactive plot

- Adding `plotOutput("plot")` in the `mainPanel()` ui function we get:



X variable: Log.population

Y variable: PM2.5.concentration

Select Year: 2015

2. Our first shiny app

2.2. Interactive plot

- But we can make the graph even more interactive using the **plotly** package
  - It allows to have information on a data point in a tooltip on hover
- In the reactive({}) environment we should:

```r
#
   ggplot(dep_data %>% filter(Year == input$year),
          aes(x = get(input$xvar), y = get(input$yvar))) +
     geom_point(alpha = .6) +
#
#
#
     geom_smooth(method = "lm", se = F) + xlab(input$xvar) + ylab(input$yvar)
#
#
```

2. Our first shiny app

2.2. Interactive plot

- But we can make the graph even more interactive using the **plotly** package
    - It allows to have information on a data point in a tooltip on hover

- In the reactive({}) environment we should:
    1. Put the ggplot in the ggplotly() function

```r
ggplotly(
   ggplot(dep_data %>% filter(Year == input$year),
          aes(x = get(input$xvar), y = get(input$yvar))) +
     geom_point(alpha = .6) +
#
#
#
     geom_smooth(method = "lm", se = F) + xlab(input$xvar) + ylab(input$yvar)
#
)
```

2. Our first shiny app

2.2. Interactive plot

- But we can make the graph even more interactive using the **plotly** package
  - It allows to have information on a data point in a tooltip on hover

- In the reactive({}) environment we should:
  1. Put the ggplot in the ggplotly() function
  2. Format the tooltip as the 'text' axis in aes()

```
ggplotly(
   ggplot(dep_data %>% filter(Year == input$year),
          aes(x = get(input$xvar), y = get(input$yvar))) +
     geom_point(alpha = .6,
                aes(text = paste0(Department, "<br>",
                                  input$xvar, ": ", round(get(input$xvar), 2), "<br>",
                                  input$yvar, ": ", round(get(input$yvar), 2)))) +
     geom_smooth(method = "lm", se = F) + xlab(input$xvar) + ylab(input$yvar)
#
)
```

2. Our first shiny app

2.2. Interactive plot

- But we can make the graph even more interactive using the **plotly** package
  - It allows to have information on a data point in a tooltip on hover

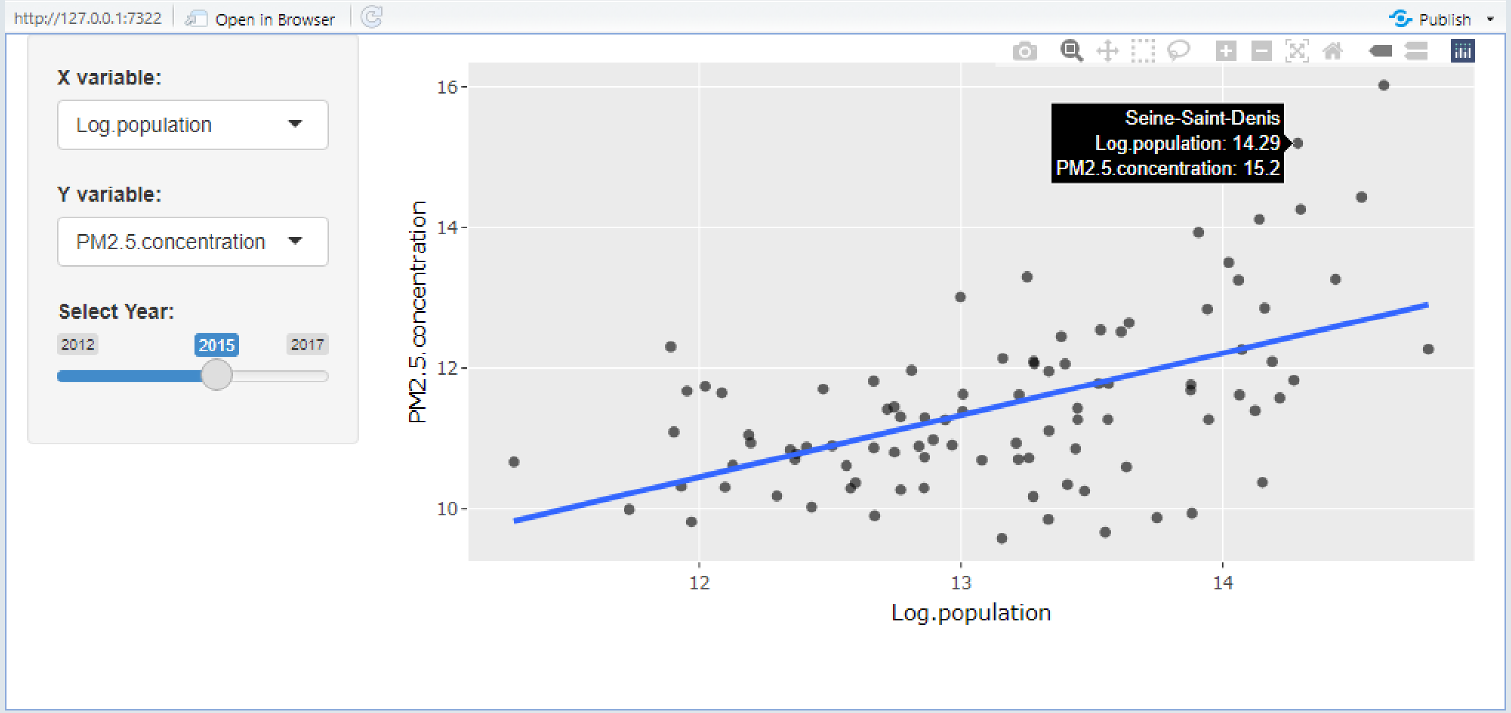
- In the reactive({}) environment we should:
  1. Put the ggplot in the ggplotly() function
  2. Format the tooltip as the 'text' axis in aes()
  3. Assign the text axis to the tooltip argument of ggplotly()

```
ggplotly(
   ggplot(dep_data %>% filter(Year == input$year),
          aes(x = get(input$xvar), y = get(input$yvar))) +
     geom_point(alpha = .6,
                aes(text = paste0(Department, "<br>",
                                  input$xvar, ": ", round(get(input$xvar), 2), "<br>",
                                  input$yvar, ": ", round(get(input$yvar), 2)))) +
     geom_smooth(method = "lm", se = F) + xlab(input$xvar) + ylab(input$yvar),
   tooltip = "text"
)
```

2. Our first shiny app

2.2. Interactive plot

- We also have to:
    - Replace the server renderPlot() function by **renderPlotly()**
    - Replace the UI plotOutput() function by **plotlyOutput()**



X variable: Log.population

Y variable: PM2.5.concentration

Select Year: 2015

2. Our first shiny app

2.3. Interactive regression results

- We can also include an interactive regression table
  - We should put the **stargazer()** function in a reactive({}) environment:

```
reg_table <- reactive({

  stargazer(lm(formula(paste0(c(input$yvar, input$xvar), collapse = "~")),
               dep_data %>% filter(Year == input$year)), type  = "html",
            dep.var.labels = input$yvar, keep.stat = c("n", "rsq"))

})
```

  - Then use the render function dedicated to console output: **renderPrint():**

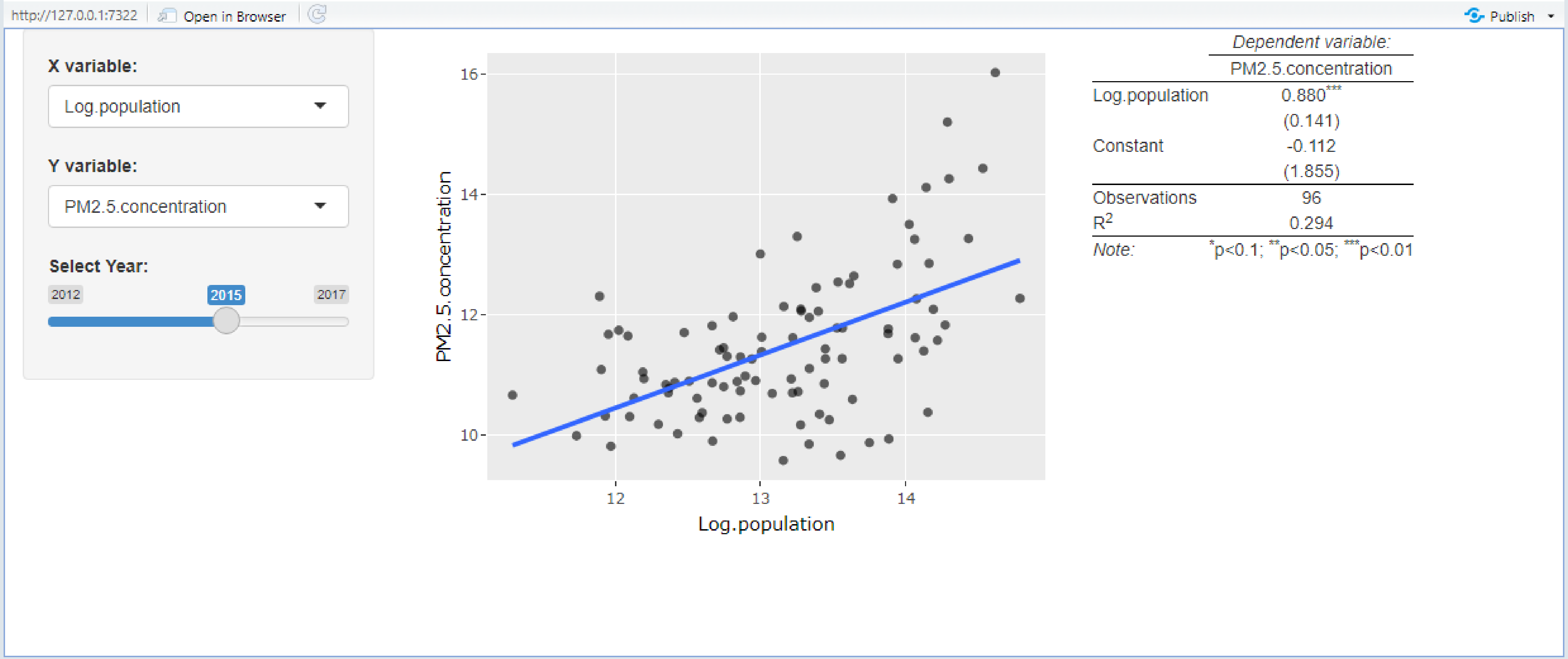
```
output$reg_table <- renderPrint({reg_table()})
```

  - And use the **htmlOuput()** UI output function in a column layout to put it side to side with the plot:

```
mainPanel(width = 9, column(7, plotlyOutput("plot")), column(5, htmlOutput("reg_table")))
```

2. Our first shiny app

2.3. Interactive regression results



X variable: Log.population

Y variable: PM2.5.concentration

Select Year: 2012 — 2015 — 2017

	Dependent variable:
	PM2.5.concentration
Log.population	0.880***
	(0.141)
Constant	-0.112
	(1.855)
Observations	96
R^2	0.294
Note:	*p<0.1; **p<0.05; ***p<0.01

Overview

1. Introduction to shiny apps ✓

- 1.1. General structure
- 1.2. User interface
- 1.3. Server
- 1.4. Layout

2. Our first shiny app ✓

- 2.1. Import data in Shiny
- 2.2. Interactive plot
- 2.3. Interactive regression results

3. More advanced tools

- 3.1. Input randomization
- 3.2. HTML formatting

4. Wrap up!

Overview

1. Introduction to shiny apps ✓

- 1.1. General structure
- 1.2. User interface
- 1.3. Server
- 1.4. Layout

2. Our first shiny app ✓

- 2.1. Import data in Shiny
- 2.2. Interactive plot
- 2.3. Interactive regression results

3. More advanced tools

- 3.1. Input randomization
- 3.2. HTML formatting

3. More advanced tools

3.1. Input randomization

➜ A nice feature would be a button to **randomly select** the x and y **variables** and the year

- Adding a button is easy, we can simply add an `actionButton()` widget in the control panel:

```r
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(...,
      actionButton(inputId = "random", label = "Random selection")
    ),
    mainPanel(...)
  )
)
```

- But what should happen in the server when clicking on it is different from what we're used to:
  - We do not want a reactive output to place in the UI
  - We want the selected inputs to change

➜ This is why we're gonna use observeEvent() instead of reactive()

3. More advanced tools

3.1. Input randomization

- The arguments of **observeEvent()** are:
  - The **id** of the input that should trigger the actions
  - The **actions** to take when the input is triggered

```
observeEvent(input$random, {
    # SELECT RANDOM INPUTS
})
```

- The action we want is to change the status of the input widgets
  - This can be done with functions of the form **update[SelectInput/SliderTextInput/...]()**
  - The first argument should be "session"
  - And the following arguments are those of the widget that we can update

```
updateSelectInput(session, inputId = "xvar", label = "X variable:",
                  choices = depvars, selected = sample(depvars, 1))
```

`sample(depvars, 1)` picks 1 variable name randomly from the vector `depvars`

3. More advanced tools

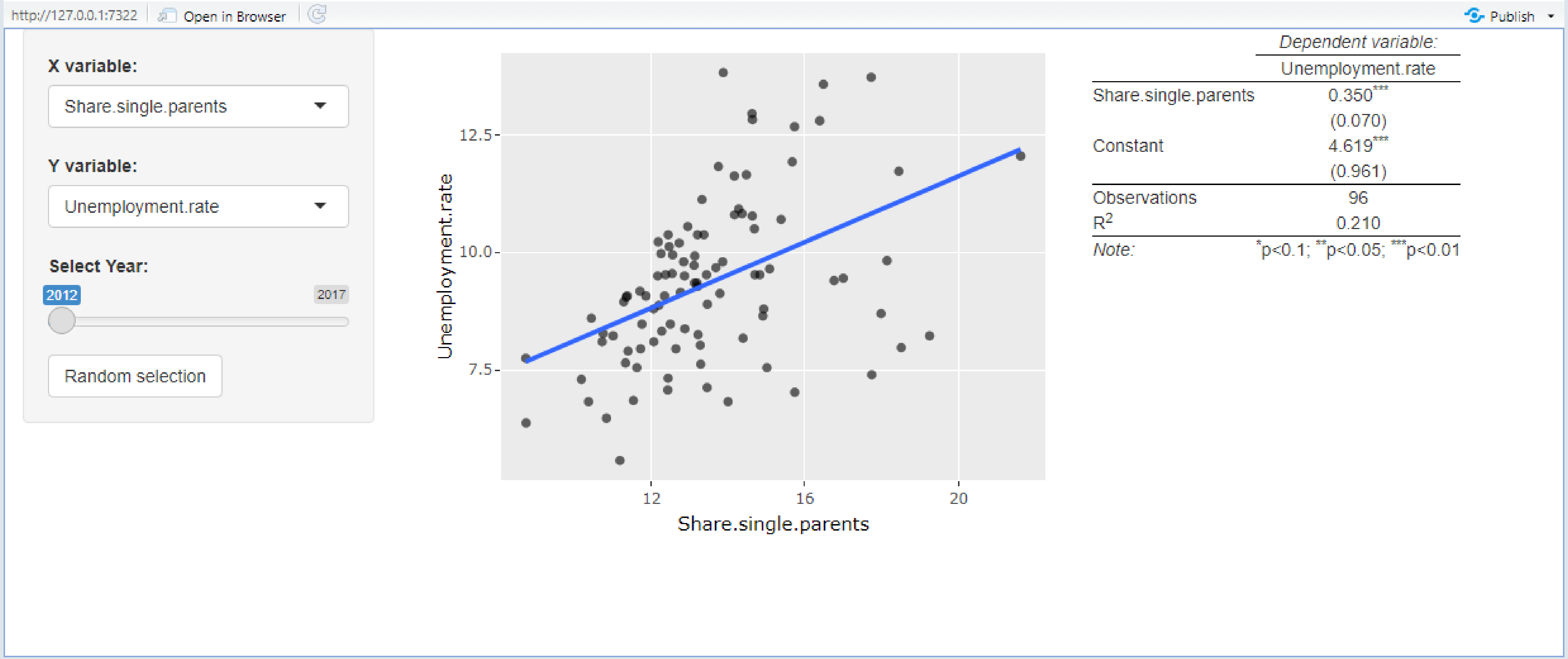
3.1. Input randomization

- For it to work, the `session` argument should also be added to the `server()` function

```
server <- function(input, output, session) {

  ...

  observeEvent(input$random, {

    updateSelectInput(session, inputId = "xvar", label = "X variable:",
                      choices = depvars, selected = sample(depvars, 1))

    updateSelectInput(session, inputId = "yvar", label = "Y variable:",
                      choices = depvars, selected = sample(depvars, 1))

    updateSliderTextInput(session, inputId = "year", label = "Select Year:",
                          choices = 2012:2017, selected = sample(2012:2017, 1))

  })

}
```

3. More advanced tools

3.1. Input randomization



X variable: Share.single.parents

Y variable: Unemployment.rate

Select Year: 2012 (range 2012–2017)

Random selection

	Dependent variable:
	Unemployment.rate
Share.single.parents	0.350***
	(0.070)
Constant	4.619***
	(0.961)
Observations	96
R^2	0.210
Note:	*p<0.1; **p<0.05; ***p<0.01

3. More advanced tools

3.2. HTML formatting

- We can apply the final touch to our application by including some **html formatting**
  - To include html in the UI we can simply use the **HTML()** function

Some html tags

```
HTML("<b>Text</b>")
```

Text

```
HTML("<i>Text</i>")
```

Text

```
HTML("<center>Text</center>")
```

Text

```
HTML("<h1>Text</h1>")
```

Text

```
HTML("<h3>Text</h3>")
```

Text

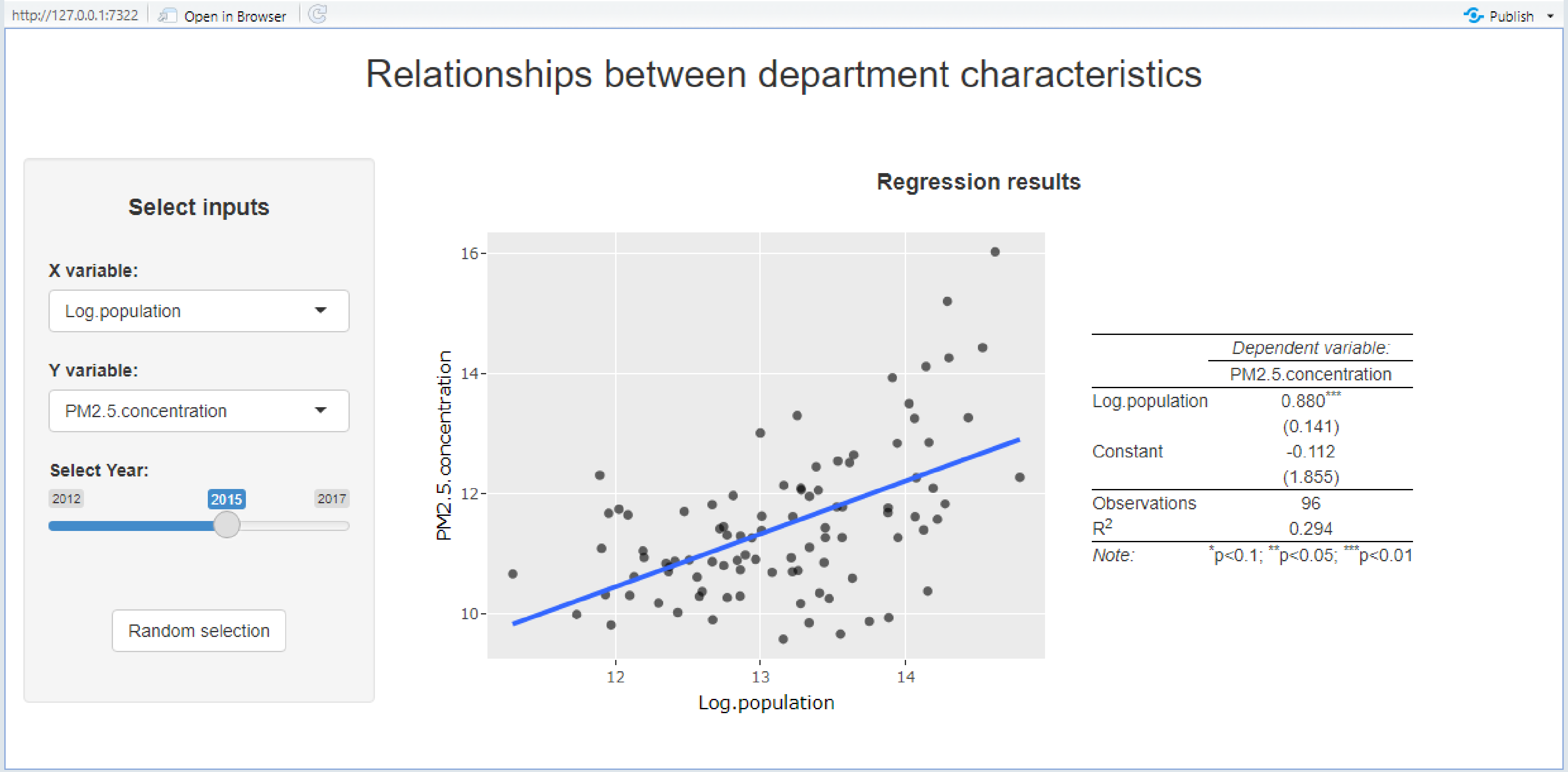
3. More advanced tools

3.2. HTML formatting

```
ui <- fluidPage(

  HTML("<center><h2>Relationships between department characteristics</h2></center><br><br>"),

  sidebarLayout(
    sidebarPanel(width = 3,

      HTML("<center><h4><b>Select inputs</b></h4></center><br>"),

      selectInput(...), selectInput(...), sliderTextInput(...),

      HTML("<br><br><center>"), actionButton(...), HTML("</center><br>")
    ),

    mainPanel(width = 9,
              HTML("<center><h4><b>Regression results</b></h4></center>"),
              column(width = 7, plotlyOutput("plot")),
              column(width = 5, HTML("<br><br><br><br><br>"), htmlOutput("reg_table")))
  )
)
```

3. More advanced tools

3.2. HTML formatting



Relationships between department characteristics

Select inputs

X variable: Log.population

Y variable: PM2.5.concentration

Select Year: 2015

Regression results

	Dependent variable:
	PM2.5.concentration
Log.population	0.880^{***}
	(0.141)
Constant	-0.112
	(1.855)
Observations	96
R^2	0.294
Note:	*p<0.1; **p<0.05; ***p<0.01

Overview

1. Introduction to shiny apps ✓

- 1.1. General structure
- 1.2. User interface
- 1.3. Server
- 1.4. Layout

2. Our first shiny app ✓

- 2.1. Import data in Shiny
- 2.2. Interactive plot
- 2.3. Interactive regression results

3. More advanced tools ✓

- 3.1. Input randomization
- 3.2. HTML formatting

4. Wrap up!

4. Wrap up!

General structure

A shiny app is composed of

A **user interface** function

- It is what is displayed to the user
    -
    -

```r
library(shiny)

ui <- fluidPage(

  #
  #
  #
  #
  #

  #

)
```

A **server** function

- It is what should be computed in the background
    -
    -

```r
server <- function(input, output) {

  #
  #
  #

  #

}

shinyApp(ui = ui, server = server)

# The file should be named app.R
```

4. Wrap up!

General structure

A shiny app is composed of

A **user interface** function

- It is what is displayed to the user, including:
    - **Input widgets**
    -

A **server** function

- It is what should be computed in the background
    -
    -

```r
library(shiny)

ui <- fluidPage(

  checkboxGroupInput(
    inputId = "boxes",
    label = "Boxes to check",
    choices = c("A", "B", "C"),
    selected = "B"),

  #

)
```

```r
server <- function(input, output) {

  #
  #
  #

  #

}

shinyApp(ui = ui, server = server)

# The file should be named app.R
```

4. Wrap up!

General structure

A shiny app is composed of

A **user interface** function

- It is what is displayed to the user, including:
  - **Input widgets**
  -

```r
library(shiny)

ui <- fluidPage(

  checkboxGroupInput(
    inputId = "boxes",
    label = "Boxes to check",
    choices = c("A", "B", "C"),
    selected = "B"),

  #

)
```

A **server** function

- It is what should be computed in the background:
  - Update inputs with **reactive({})**
  -

```r
server <- function(input, output) {

  react_tb <- reactive({
    tibble(selected = input$boxes)
  })

  #

}

shinyApp(ui = ui, server = server)

# The file should be named app.R
```

4. Wrap up!

General structure

A shiny app is composed of

A **user interface** function

- It is what is displayed to the user, including:
  - **Input widgets**
  -

A **server** function

- It is what should be computed in the background:
  - Update inputs with **reactive({})**
  - Render output with **render[Table/Plot/...]()**

```r
library(shiny)

ui <- fluidPage(

  checkboxGroupInput(
    inputId = "boxes",
    label = "Boxes to check",
    choices = c("A", "B", "C"),
    selected = "B"),

  #

)
```

```r
server <- function(input, output) {

  react_tb <- reactive({
    tibble(selected = input$boxes)
  })

  output$table <- renderTable({react_tb()})

}

shinyApp(ui = ui, server = server)

# The file should be named app.R
```

4. Wrap up!

General structure

A shiny app is composed of

A **user interface** function

- It is what is displayed to the user, including:
  - **Input widgets**
  - **Reactive outputs**

A **server** function

- It is what should be computed in the background:
  - Update inputs with **reactive({})**
  - Render output with **render[Table/Plot/...]()**

```
library(shiny)

ui <- fluidPage(

  checkboxGroupInput(
    inputId = "boxes",
    label = "Boxes to check",
    choices = c("A", "B", "C"),
    selected = "B"),

  tableOutput("table")

)
```

```
server <- function(input, output) {

  react_tb <- reactive({
    tibble(selected = input$boxes)
  })

  output$table <- renderTable({react_tb()})

}

shinyApp(ui = ui, server = server)

# The file should be named app.R
```

4. Wrap up!

```r
library(shiny)
#
#

ui <- fluidPage(
#
#
#
#
#
)

server <- function(input, output) {
#
#
#
#
#
}

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

4. Wrap up!

```r
library(shiny)
#
data(iris)

ui <- fluidPage(
  selectInput(inputId = "x", label = "Select X variable:",
              choices = names(iris), selected = names(iris)[1]),
  selectInput(inputId = "y", label = "Select Y variable:",
              choices = names(iris), selected = names(iris)[2]),
#
)

server <- function(input, output) {
#
#
#
#
#
}

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

4. Wrap up!

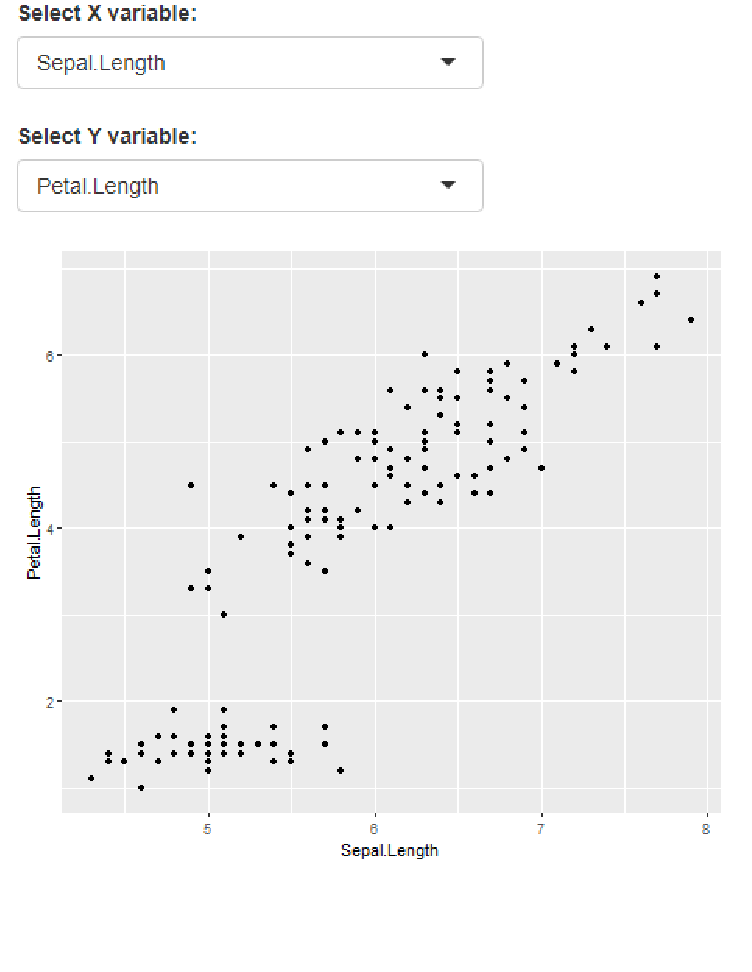
```r
library(shiny)
library(tidyverse)
data(iris)

ui <- fluidPage(
  selectInput(inputId = "x", label = "Select X variable:",
              choices = names(iris), selected = names(iris)[1]),
  selectInput(inputId = "y", label = "Select Y variable:",
              choices = names(iris), selected = names(iris)[2]),
#
)

server <- function(input, output) {
  reactive_plot <- reactive({
    ggplot(iris, aes(x = get(input$x), y = get(input$y))) +
      geom_point() + xlab(input$x) + ylab(input$y)
  })
#
}

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

4. Wrap up!

```
library(shiny)
library(tidyverse)
data(iris)

ui <- fluidPage(
  selectInput(inputId = "x", label = "Select X variable:",
              choices = names(iris), selected = names(iris)[1]),
  selectInput(inputId = "y", label = "Select Y variable:",
              choices = names(iris), selected = names(iris)[2]),
#
)

server <- function(input, output) {
  reactive_plot <- reactive({
    ggplot(iris, aes(x = get(input$x), y = get(input$y))) +
      geom_point() + xlab(input$x) + ylab(input$y)
  })
  output$plot <- renderPlot({reactive_plot()})
}

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

4. Wrap up!

```r
library(shiny)
library(tidyverse)
data(iris)

ui <- fluidPage(
  selectInput(inputId = "x", label = "Select X variable:",
              choices = names(iris), selected = names(iris)[1]),
  selectInput(inputId = "y", label = "Select Y variable:",
              choices = names(iris), selected = names(iris)[2]),
  plotOutput("plot")
)

server <- function(input, output) {
  reactive_plot <- reactive({
    ggplot(iris, aes(x = get(input$x), y = get(input$y))) +
      geom_point() + xlab(input$x) + ylab(input$y)
  })
  output$plot <- renderPlot({reactive_plot()})
}

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



Select X variable: Sepal.Length

Select Y variable: Petal.Length