

Developing Data Products Course Notes

Xing Su

Contents

Shiny (tutorial)	3
Structure of Shiny App	3
ui.R	3
ui.R Example	5
server.R	7
server.R Example	8
Distributing Shiny Application	8
Debugging	8
manipulate Package	9
Example	9
rCharts	10
Example	10
GoogleVis API	12
Example (line chart)	13
Example (merging graphs)	14
ShinyApps.io (link)	15
plot.ly (link)	15
Example	15
Structure of a Data Analysis Report	17
Slidify	18
YAML (YAML Ain't Markup Language/Yet Another Markup Language)	18
Slides	19
Publishing	20
RStudio Presentation	21
Creating Presentation	21
Slidify vs RStudio Presenter	22
R Package	23
R Package Components	23
DESCRIPTION file	24
R Code	24
NAMESPACE file	24

Documentation	25
Building/Checking Package	26
Checklist for Creating Package	27
Example: <code>topten</code> function	27
R Classes and Methods	29
Objected Oriented Programming in R	29
Creating a New Class/Methods	32
Yhat (link)	34
Deploying the Model	35
Accessing the Model	35

Shiny ([tutorial](#))

- Shiny = platform for creating interactive R program embedded on web pages (made by RStudio)
- knowledge of these helpful: HTML (web page structure), css (style), JavaScript (interactivity)
- **OpenCPU** = project created by Jerom Ooms providing API for creating more complex R/web apps
- `install.packages("shiny"); library(shiny)` = install/load shiny package
- capabilities
 - upload or download files
 - tabbed main panels
 - editable data tables
 - dynamic UI
 - user defined inputs/outputs
 - submit button to control when calculations/tasks are processed

Structure of Shiny App

- **two** scripts (in one directory) make up a Shiny project
 - `ui.R` - controls appearance/all style elements
 - * alternatively, a `www` directory with an `index.html` file enclosed can be used instead of `ui.R`
 - **Note:** *output is rendered into HTML elements based on matching their id attribute to an output slot and by specifying the requisite css class for the element (in this case either `shiny-text-output`, `shiny-plot-output`, or `shiny-html-output`)*
 - it is possible to create highly customized user-interfaces using user-defined HTML/CSS/JavaScript
 - `server.R` - controls functions
- `runApp()` executes the Shiny application
 - `runApp(display.mode = 'showcase')` = displays the code from `ui.R` and `server.R` and highlights what is being executed depending on the inputs
- **Note:** `", "` must be included **ONLY INBETWEEN** objects/functions on the same level

ui.R

- `library(shiny)` = first line, loads the shiny package
- `shinyUI()` = shiny UI wrapper, contains sub-methods to create panels/parts/viewable object
- `pageWithSideBar()` = creates page with main/side bar division
- `headerPanel("title")` = specifies header of the page
- `sideBarPanel()` = specifies parameters/objects in the side bar (on the **left**)
- `mainPanel()` = specifies parameters/objects in the main panel (on the **right**)
- for better control over style, use `shinyUI(fluidpage())` ([tutorial](#)) <- produces responsive web pages
 - `fluidRow()` = creates row of content with width 12 that can be subdivided into columns
 - * `column(4, ...)` = creates a column of width 4 within the fluid row
 - * `style = "CSS"` = can be used as the last element of the column to specify additional style
- `absolutePanel(top=0, left=0, right=0)` = used to produce floating panels on top of the page ([documentation](#))
 - `fixed = TRUE` = panel will not scroll with page, which means the panel will always stay in the same position as you scroll through the page

- `draggable = TRUE` = make panel movable by the user
 - `top = 40 / bottom = 50` = position from the top/bottom edge of the browser window
 - * `top = 0, bottom = 0` = creates panel that spans the entire vertical length of window
 - `left = 40 / right = 50` = position from the left/right edge of the browser window
 - * `top = 0, bottom = 0` = creates panel that spans the entire horizontal length of window
 - `height = 30 / width = 40` = specifies the height/width of the panel
 - `style = "opacity:0.92; z-index = 100"` = makes panel transparent and ensures the panel is always the top-most element
- **content objects/functions**
 - ***Note:** more HTML tags can be found [here](#)*
 - ***Note:** most of the content objects (`h1`, `p`, `code`, etc) can use **both** double and single quotes to specify values, just be careful to be consistent*
 - `h1/2/3/4('heading')` = creates heading for the panel
 - `p('paragraph')` = creates regular text/paragraph
 - `code('code')` = renders code format on the page
 - `br()` = inserts line break
 - `tags$hr()` = inserts horizontal line
 - `tags$ol()/tags$ul()` = initiates ordered/unordered list
 - `div(... , style = "CSS Code") / span(... , style = "CSS Code")` = used to add additional style to particular parts of the app
 - * `div` should be used for a section/block, `span` should be used for a specific part/inline
 - `withMathJax()` = add this element to allow Shiny to process LaTeX
 - * inline LaTeX must be wrapped like this: `\\(LaTeX\\)`
 - * block equations are still wrapped by: `$$LaTeX$$`
 - **inputs**
 - `textInput(inputId = "id", label = "textLabel")` = creates a plain text input field
 - * `inputId` = field identifier
 - * `label` = text that appear above/before a field
 - `numericInput('HTMLLabel', 'printedLabel', value = 0, min = 0, max = 10, step = 1)` = create a number input field with incrementer (up/down arrows)
 - * `'HTMLLabel'` = name given to the field, not printed, and can be called
 - * `'printedLabel'` = text that shows up above the input box explaining the field
 - * `value` = default numeric value that the field should take; 0 is an example
 - * `min` = minimum value that can be set in the field (if a smaller value is manually entered, then the value becomes the minimum specified once user clicks away from the field)
 - * `max` = max value that can be set in the field
 - * `step` = increments for the up/down arrows
 - * more arguments can be found in `?numericInput`
 - `checkboxGroupInput("id2", "Checkbox", choices = c("Value 1" = "1", ...), selected = "1", inline = TRUE)` = creates a series of checkboxes
 - * `"id2", "Checkbox"` = field identifier/label
 - * `choices` = list of checkboxes and their labels
 - `format = "checkboxName" = "fieldIdentifier"`
 - ***Note:** `fieldIdentifier` should generally be different from checkbox to checkbox, so we can properly identify the responses*
 - * `selected` = specifies the checkboxes that should be selected by default; uses `fieldIdentifier` values

- * inline = whether the options should be displayed inline
- `dateInput("fieldID", "fieldLabel")` = creates a selectable date field (dropdown calendar/date picker automatically generated)
 - * "fieldID" = field identifier
 - * "fieldLabel" = text/name displayed above fields
 - * more arguments can be found in `?dateInput`
- `submitButton("Submit")` = creates a submit button that updates the output/calculations only when the user submits the new inputs (default behavior = all changes update reactively/in real time)
- `actionButton(inputId = "goButton", label = "test")` = creates a button with the specified label and id
 - * output can be specified for when the button is clicked
- `sliderInput("id", "label", value = 70, min = 62, max = 74, 0.05)` = creates a slider for input
 - * arguments similar to `numericInput` and more information can be found `?sliderInput`

• outputs

- ***Note:** every variable called here must have a corresponding method corresponding method from the `output` element in `server.R` to render their value*
- `textOutput("fieldId", inline = FALSE)` = prints the value of the variable/field in text format
 - * inline = TRUE = inserts the result inline with the HTML element
 - * inline = FALSE = inserts the result in block code format
- `verbatimTextOutput("fieldId")` = prints out the value of the specified field defined in `server.R`
- `plotOutput('fieldId')` = plots the output ('sampleHist' for example) created from `server.R` script
- `output$test <- renderText({input$goButton}); isolate(paste(input$t1, input$2))}`
= isolate action executes when the button is pressed
 - * `if (input$goButton == 1){ Conditional statements }` = create different behavior depending on the number of times the button is pressed

ui.R Example

- below is part of the ui.R code for a project on Shiny

```
# load shiny package
library(shiny)
# begin shiny UI
shinyUI(navbarPage("Shiny Project",
  # create first tab
  tabPanel("Documentation",
    # load MathJax library so LaTeX can be used for math equations
    withMathJax(), h3("Why is the Variance Estimator  $S^2$  divided by  $(n-1)$ "),
    # paragraph and bold text
    p("The ", strong("sample variance"), " can be calculated in ", strong(em("two")),
      " different ways:",
      "$$S^2 \text{ \texttt{\mbox{(unbiased)}}} = \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n-1} \\
      \text{ \texttt{\mbox{and}}} S^2 \text{ \texttt{\mbox{(biased)}}} = \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n}$$",
      "The unbiased calculation is most often used, as it provides a ",
      strong(em("more accurate")), " estimate of population variance"),
    # break used to space sections
    br(), p("To show this empirically, we simulated the following in the ",
```

```

        strong("Simulation Experiment"), " tab: "), br(),
# ordered list
tags$ol(
  tags$li("Create population by drawing observations from values 1 to 20."),
  tags$li("Draw a number of samples of specified size from the population"),
  tags$li("Plot difference between sample and true population variance"),
  tags$li("Show the effects of sample size vs accuracy of variance estimated")
)),
# second tab
tabPanel("Simulation Experiment",
  # fluid row for space holders
  fluidRow(
    # fluid columns
    column(4, div(style = "height: 150px")),
    column(4, div(style = "height: 150px")),
    column(4, div(style = "height: 150px")),
    # main content
    fluidRow(
      column(12, h4("We start by generating a population of ",
        span(textOutput("population", inline = TRUE),
          style = "color: red; font-size: 20px"),
          " observations from values 1 to 20:"),
        tags$hr(), htmlOutput("popHist"),
        # additional style
        style = "padding-left: 20px"
      )
    ),
    # absolute panel
    absolutePanel(
      # position attributes
      top = 50, left = 0, right = 0,
      fixed = TRUE,
      # panel with predefined background
      wellPanel(
        fluidRow(
          # sliders
          column(4, sliderInput("population", "Size of Population:",
            min = 100, max = 500, value = 250),
            p(strong("Population Variance: "),
              textOutput("popVar", inline = TRUE))),
          column(4, sliderInput("numSample", "Number of Samples:",
            min = 100, max = 500, value = 300),
            p(strong("Sample Variance (biased): "),
              textOutput("biaVar", inline = TRUE))),
          column(4, sliderInput("sampleSize", "Size of Samples:",
            min = 2, max = 15, value = 10),
            p(strong("Sample Variance (unbiased): "),
              textOutput("unbiaVar", inline = TRUE))))),
          style = "opacity: 0.92; z-index: 100;"
        ))
      )
    )
  )
)

```

server.R

- preamble/code to set up environment (executed only *once*)
 - start with `library()` calls to load packages/data
 - define/initiate variables and relevant default values
 - * `<<-` operator should be used to assign values to variables in the parent environment
 - * `x <<- x + 1` will define `x` to be the sum of 1 and the value of `x` (defined in the parent environment/working environment)
 - any other code that you would like to only run once
- `shinyServer()` = initiates the server function
 - `function(input, output){}` = defines a function that performs actions on the inputs user makes and produces an output object
 - **non-reactive** statements/code will be executed *once for each page refresh/submit*
 - **reactive** functions/code are *run repeatedly* as values are updated (i.e. render)
 - * *Note: Shiny only runs what is needed for reactive statements, in other words, the rest of the code is left alone*
 - * `reactive(function)` = can be used to wrap functions/expressions to create reactive expressions
 - `renderText({x()})` = returns value of `x`, “`()`” must be included (syntax)
- reactive function example

```
# start shinyServer
shinyServer(
# specify input/output function
function(input, output) {
  # set x as a reactive function that adds 100 to input1
  x <- reactive({as.numeric(input$text1)+100})
  # set value of x to output object text1
  output$text1 <- renderText({x()})
  # set value of x plus value of input object text2 to output object text1
  output$text2 <- renderText({x() + as.numeric(input$text2)})
}
)
```

- functions/output objects in `shinyServer()`
 - `output$oid1 <- renderPrint({input$id1})` = stores the user input value in field `id1` and stores the rendered, printed text in the `oid1` variable of the output object
 - * `renderPrint({expression})` = reactive function to render the specified expression
 - * `{}` is used to ensure the value is an expression
 - * `oid1` = variable in the output object that stores the result from the subsequent command
 - `output$sampleHist <- renderPlot({code})` = stores plot generated by code into `sampleHist` variable
 - * `renderPlot({code})` = renders a plot generated by the enclosed R code
 - `output$sampleGVisPlot <- renderGvis({code})` = renders Google Visualization object

server.R Example

- below is part of the server.R code for a project on Shiny that uses googleVis

```
# load libraries
library(shiny)
require(googleVis)
# begin shiny server
shinyServer(function(input, output) {
  # define reactive parameters
  pop<- reactive({sample(1:20, input$population, replace = TRUE)})
  bootstrapSample<-reactive({sample(pop(),input$sampleSize*input$numSample,
    replace = TRUE)})
  popVar<- reactive({round(var(pop()),2)})
  # print text through reactive funtion
  output$biaVar <- renderText({
    sample<- as.data.frame(matrix(bootstrapSample(), nrow = input$numSample,
      ncol =input$sampleSize))
    return(round(mean(rowSums((sample-rowMeans(sample))^2)/input$sampleSize), 2))
  })
  # google visualization histogram
  output$popHist <- renderGvis({
    popHist <- gvisHistogram(data.frame(pop()), options = list(
      height = "300px",
      legend = "{position: 'none'}", title = "Population Distribution",
      subtitle = "samples randomly drawn (with replacement) from values 1 to 20",
      histogram = "{ hideBucketItems: true, bucketSize: 2 }",
      hAxis = "{ title: 'Values', maxAlternation: 1, showTextEvery: 1}",
      vAxis = "{ title: 'Frequency'}"
    ))
    return(popHist)
  })
})
```

Distributing Shiny Application

- running code locally = running local server and browser routing through local host
- quickest way = send application directory
- possible to create R package and create a wrapper that calls `runApp` (requires R knowledge)
- another option = run shiny server ([link](#))

Debugging

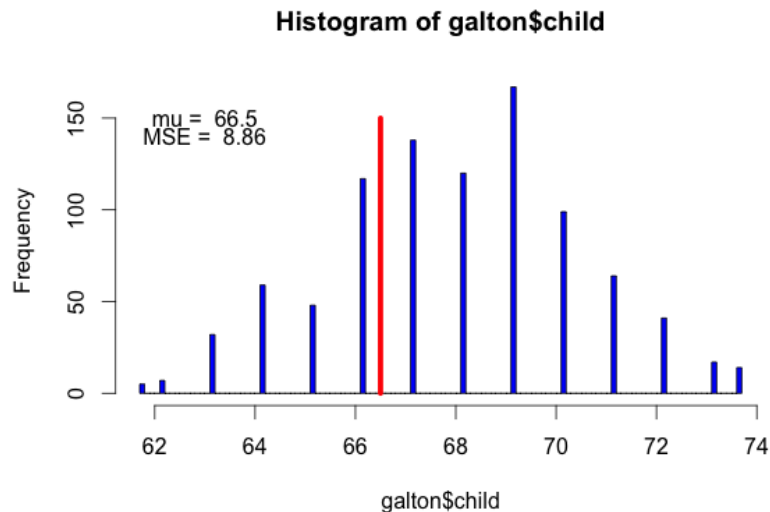
- `runApp(display.mode = 'showcase')` = highlights execution while running a shiny application
- `cat` = can be used to display output to stdout/R console
- `browser()` = interrupts execution ([tutorial](#))

manipulate Package

- `manipulate` = package/function can be leveraged to create quick interactive graphics by allowing the user to vary the different variables to a model/calculation
- creates sliders/checkbox/picker for the user ([documentation](#))

Example

```
# load data and manipulate package
library(UsingR)
library(manipulate)
# plotting function
myHist <- function(mu){
  # histogram
  hist(galton$child,col="blue",breaks=100)
  # vertical line to highlight the mean
  lines(c(mu, mu), c(0, 150),col="red",lwd=5)
  # calculate mean squared error
  mse <- mean((galton$child - mu)^2)
  # updates the mean value as the mean is changed by the user
  text(63, 150, paste("mu = ", mu))
  # updates the mean squared error value as the mean is changed by the user
  text(63, 140, paste("MSE = ", round(mse, 2)))
}
# creates a slider to vary the mean for the histogram
manipulate(myHist(mu), mu = slider(62, 74, step = 0.5))
```

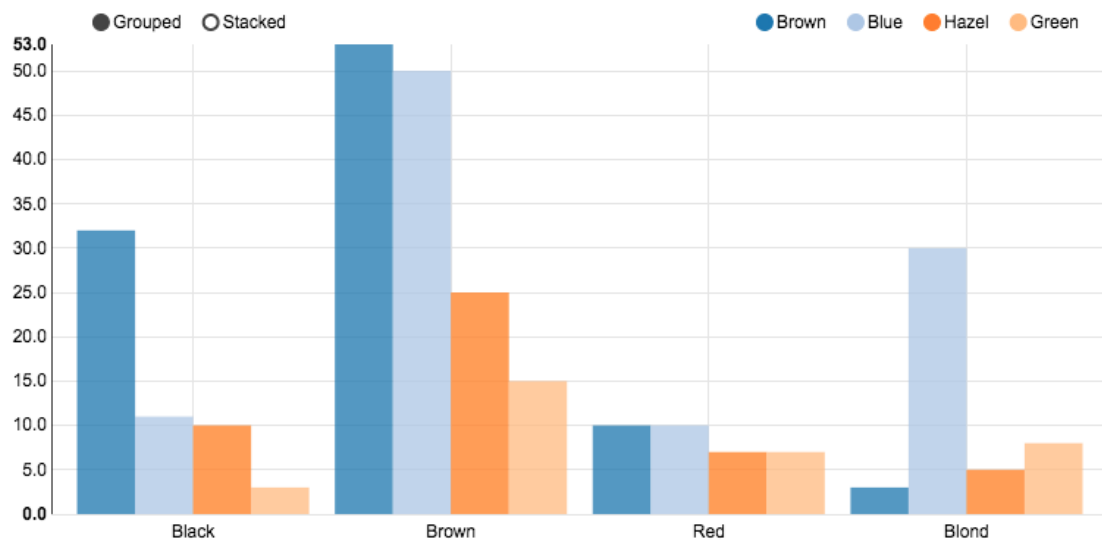


rCharts

- rCharts = simple way of creating interactive JavaScript visualization using R
 - more in-depth knowledge of D3 will be needed to create more complex tools
 - written by Ramnath Vaidyanathan
 - uses formula interface to specify plots (like the `lattice` plotting system)
 - displays interactive tool tips when hovering over data points on the plots
- installation
 - devtools must be installed first (`install.packages("devtools")`)
 - `require(devtools); install_github('rCharts', 'ramnathv')` installs the rCharts package from GitHub
- plot types
 - ***Note:** each of the following JS library has different default styles as well as a multitude of capabilities; the following list is just what was demonstrated and more documentation can be found in the corresponding links*
 - [Polychart js library] `rPlot` -> paneled scatter plots
 - [Morris js library] `mPlot` -> time series plot (similar to stock price charts)
 - [NVD3 js library] `nPlot` -> stacked/grouped bar charts
 - [xCharts js library] -> shaded line graphs
 - [HighChart js library] -> stacked (different styles) scatter/line charts
 - [LeafLet js library] -> interactive maps
 - [Rickshaw js library] -> stacked area plots/time series
- rChart objects have various attributes/functions that you can use when saved `n1 <- nplot(...)`
 - `n1$ + TAB` in R Console brings up list of all functions contained in the object
 - `n1$html()` = prints out the HTML for the plot
 - `n1$save(filename)` = embeds code into slidify document
 - `n1$print()` = print out the JavaScript
- `n1$show("inline", include_assets = TRUE, cdn = F)` = embed HTML/JS code directly with in Rmd file (for HTML output)
 - `n1$publish('plotname', host = 'gist'/'rpubs')` = publishes the plot under the specified plotname as a gist or to rpubs
- to use with *slidify*, the following YAML (Yet Another Markup Language/YAML Ain't Markup Language) must be added
 - `yaml ext_widgets : {rCharts: ["libraries/nvd3"]}`

Example

```
# load rCharts package
require(rCharts); library(datasets); library(knitr)
# create dataframe with HairEyeColor data
haireye = as.data.frame(HairEyeColor)
# create a nPlot object
n1 <- nPlot(Freq ~ Hair, group = 'Eye', type = 'multiBarChart',
            data = subset(haireye, Sex == 'Male'))
# save the nPlot object to a html page
n1$show("inline", include_assets = TRUE, cdn = F)
```

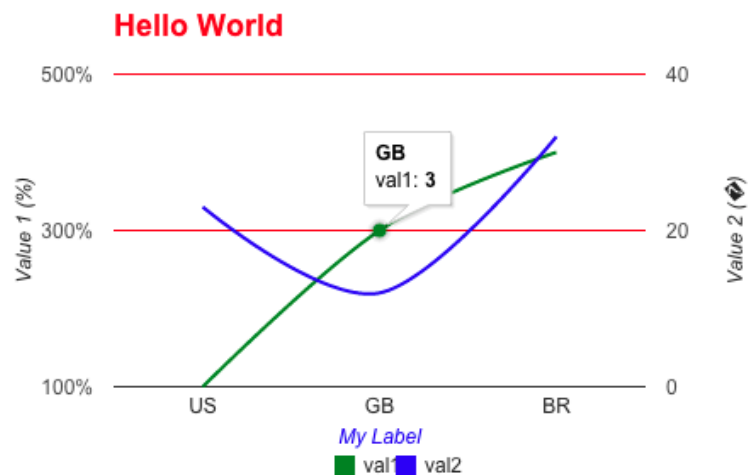


GoogleVis API

- GoogleVis allows R to create interactive HTML graphics (Google Charts)
- **chart types** (format = gvis+ChartType)
 - Motion charts: `gvisMotionChart`
 - Interactive maps: `gvisGeoChart`
 - Interactive tables: `gvisTable`
 - Line charts: `gvisLineChart`
 - Bar charts: `gvisColumnChart`
 - Tree maps: `gvisTreeMap`
 - more charts can be found [here](#)
- configuration options and default values for arguments for each of the plot types can be found [here](#)
- `print(chart, "chart")` = prints the JavaScript for creating the interactive plot so it can be embedded in slidify/HTML document
 - `print(chart)` = prints HTML + JavaScript directly
- alternatively, to print the charts on a HTML page, you can use `op <- options(gvis.plot.tag='chart')`
- this sets the googleVis options first to change the behaviour of `plot.gvis`, so that *only the chart component* of the HTML file is written into the output file
- `plot(chart)` can then be called to print the plots to HTML
- `gvisMerge(chart1, chart2, horizontal = TRUE, tableOptions = "bgcolor = \"#CCCCCC\" cellspacing = 10)` = combines the two plots into one horizontally (1 x 2 panel)
 - *Note: `gvisMerge()` can only combine **TWO** plots at a time*
 - `horizontal = FALSE` = combines plots vertically (TRUE for horizontal combination)
 - `tableOptions = ...` = used to specify attributes of the combined plot
- `demo(googleVis)` = demos how each of the plot works
- **resources**
 - [vignette](#)
 - [documentation](#)
 - [plot gallery](#)
 - [FAQ](#)

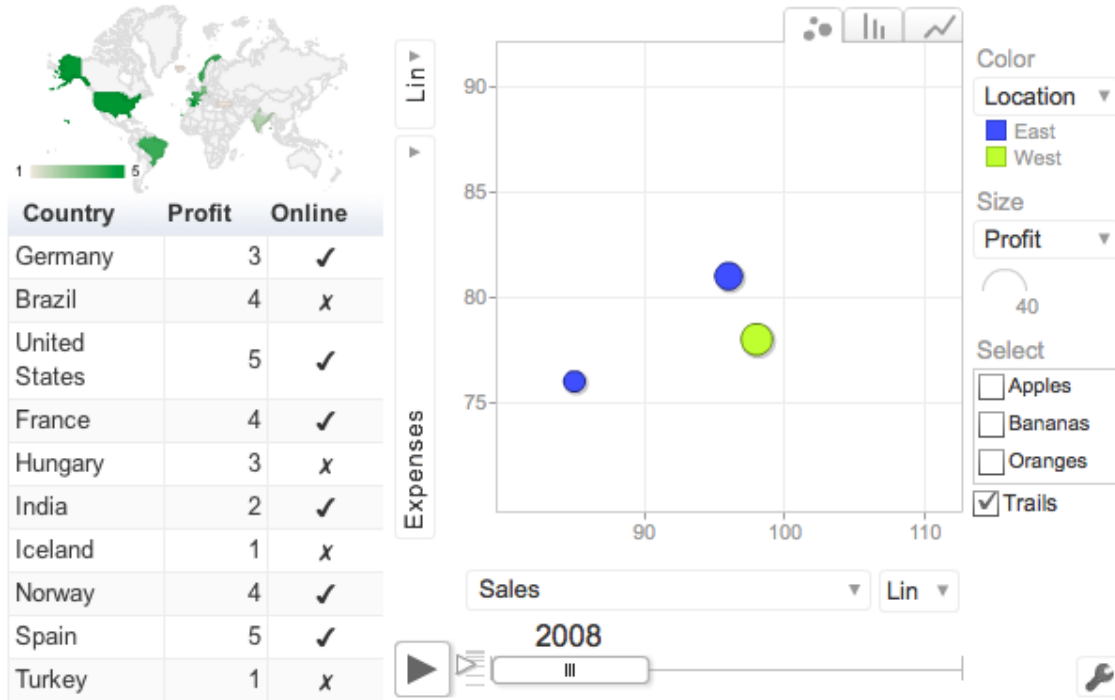
Example (line chart)

```
# load googleVis package
library(googleVis)
# set gvis.plot options to only return the chart
op <- options(gvis.plot.tag='chart')
# create initial data with x variable as "label" and y variable as "var1/var2"
df <- data.frame(label=c("US", "GB", "BR"), val1=c(1,3,4), val2=c(23,12,32))
# set up a gvisLineChart with x and y
Line <- gvisLineChart(df, xvar="label", yvar=c("val1","val2"),
  # set options for the graph (list) - title and location of legend
  options=list(title="Hello World", legend="bottom",
    # set title text style
    titleTextStyle="{color:'red', fontSize:18}",
    # set vertical gridlines
    vAxis="{gridlines:{color:'red', count:3}}",
    # set horizontal axis title and style
    hAxis="{title:'My Label', titleTextStyle:{color:'blue'}}",
    # set plotting style of the data
    series="[{color:'green', targetAxisIndex: 0},
      {color: 'blue',targetAxisIndex:1}]",
    # set vertical axis labels and formats
    vAxes="[{title:'Value 1 (%)', format:'##,#####%'},
      {title:'Value 2 (\U00A3)'}]",
    # set line plot to be smoothed and set width and height of the plot
    curveType="function", width=500, height=300
  ))
# print the chart in JavaScript
plot(Line)
```



Example (merging graphs)

```
G <- gvisGeoChart(Exports, "Country", "Profit", options=list(width=200, height=100))
T1 <- gvisTable(Exports, options=list(width=200, height=270))
M <- gvisMotionChart(Fruits, "Fruit", "Year", options=list(width=400, height=370))
GT <- gvisMerge(G, T1, horizontal=FALSE)
GTM <- gvisMerge(GT, M, horizontal=TRUE, tableOptions="bgcolor=\"#CCCCCC\" cellspacing=10")
plot(GTM)
```



- **Note:** the motion chart only displays when it is hosted on a server or a trusted Macromedia source, see [googlVis vignette](#) for more details

ShinyApps.io ([link](#))

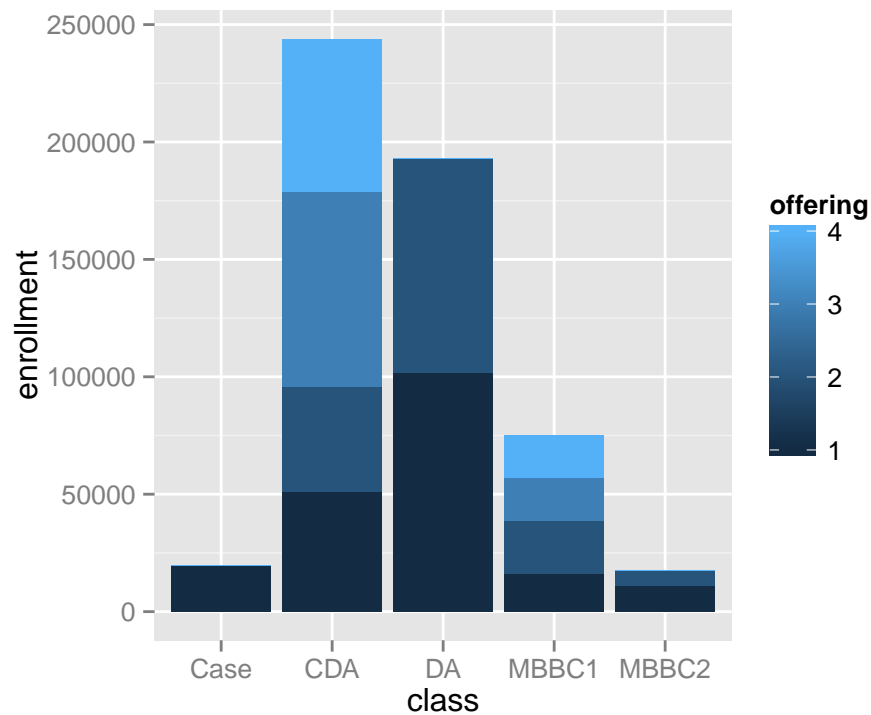
- platform created by RStudio to share Shiny apps on the web
- log in through GitHub/Google, and set up access in R
 1. Make sure you have `devtools` installed in R (`install.packages("devtools")`)
 2. enter `devtools::install_github('rstudio/shinyapps')`, which installs the `shinyapps` package from GitHub
 3. follow the instructions to authenticate your shiny apps account in R through the generated token
 4. publish your app through `deployApp()` command
- the apps your deploy will be hosted on ShinyApps.io under your account

plot.ly ([link](#))

- platform share and edit plots modularly on the web ([examples](#))
 - every part of the plot can be customized and modified
 - graphs can be converted from one language to another
- you can choose to log in through Facebook/Twitter/Google/GitHub
 1. make sure you have `devtools` installed in R
 2. enter `devtools::install_github("ropensci/plotly")`, which installs `plotly` package from GitHub
 3. go to <https://plot.ly/r/getting-started/> and follow the instructions
 4. enter `library(plotly); set_credentials_file("<username>", "<token>")` with the appropriate username and token filled in
 5. use `plotly()` methods to upload plots to your account
 6. modify any part of the plot as you like once uploaded
 7. share the plot

Example

```
# load packages
library(plotly); library(ggplot2)
# make sure your plot.ly credentials are set correctly using the following command
# set_credentials_file(username=FILL IN, api_key=FILL IN)
# load data
load("courseraData.rda")
# bar plot using ggplot2
g <- ggplot(myData, aes(y = enrollment, x = class, fill = offering))
g <- g + geom_bar(stat = "identity")
g
```



```
# initiate plotly object
py <- plotly()
# interface with plot.ly and ggplot2 to upload the plot to plot.ly under your credentials
out <- py$ggplotly(g)
# typing this in R console will return the url of the generated plot
out$response$url
```

```
## [1] "https://plot.ly/~sxing/54"
```

- the above is the response URL for the plot created on plot.ly
- **Note:** this particular URL corresponds to the plot on my personal account

Structure of a Data Analysis Report

- can be blog post or formal report for analysis of given dataset
- components
 - ***prompt*** file = analysis being performed (this file is not mandatory/available in all cases)
 - ***data*** folder = data to be analyzed + report files
 - * ***Note:*** *always keep track of data provenance (for reproducibility)*
 - ***code*** folder= rawcode vs finalcode
 - * ***rawcode*** = analysis performed (.Rmd file)
 - all exploratory data analysis
 - not for sharing with others
 - * ***finalcode*** = only analysis to be shared with other people/summarizations (.Rmd file)
 - not necessarily final but pertinent to analysis discussed in final report
 - * ***figures*** folder = final plots that have all appropriate formatting for distribution/presentation
 - * ***writing*** folder = final analysis report and final figure caption
 - report should have the following sections: title, introduction, methods, results, conclusions, references
 - final figure captions should correspond with the figures created

Slidify

- create data-centric presentations created by Ramnath Vaidyanathan
- amalgamation of knitr, Markdown, JavaScript libraries for HTML5 presentations
- easily extendable/customizable
- allows embedded code chunks and mathematical formulas (MathJax JS library) to be rendered correctly
- final products are HTML files, which can be viewed with any web browser and shared easily
- **installation**
 1. make sure you have `devtools` package installed in R
 2. enter `install_github('slidify', 'ramnathv'); install_github('slidifyLibraries', 'ramnathv')` to install the slidify packages
 3. load slidify package with `library(slidify)`
 4. set the working directory to the project you are working on with `setwd("~/project")`
- `author("title")` = sets up initial files for a new slidify project (performs the following things)
 1. `title` (or any name you typed) directory is created inside the current working directory
 2. `assets` subdirectory and a file named `index.Rmd` are created inside `title` directory
 3. `assets` subdirectory is populated with the following empty folders:
 - `css`
 - `img`
 - `js`
 - `layouts`
 - ***Note:** any custom CSS/images/JavaScript you want to use should be put into the above folders correspondingly*
 4. `index.Rmd` R Markdown file will open up in RStudio
- `slidify("index.Rmd")` = processes the R Markdown file into a HTML page and imports all necessary libraries
- `library(knitr); browseURL("index.html")` = opens up the built-in web browser in R Studio and displays the slidify presentation
 - ***Note:** this is only necessary the first time; you can refresh the page to reflect any changes after saving the HTML file*

YAML (YAML Ain't Markup Language/Yet Another Markup Language)

- used to specify options for the R Markdown/slidify at the beginning of the file
- format: `field : value # comment`
 - `title` = title of document
 - `subtitle` = subtitle of document
 - `author` = author of document
 - `job` = occupation of author (can be left blank)
 - `framework` = controls formatting, usually the name of a library is used (i.e. `io2012`)
 - * `io2012`
 - * `html5slides`
 - * `deck.js`

- * [dzslides](#)
- * [landslide](#)
- * [Slidy](#)
- **highlighter** = controls effects for presentation (i.e **highlight.js**)
- **hitheme** = specifies theme of code (i.e. **tomorrow**)
- **widgets** = loads additional libraries to display LaTeX math equations(**mathjax**), quiz-styles components (quiz), and additional style (**bootstrap** → Twitter-created style)
 - * for math expressions, the code should be enclosed in **\$expression\$** for inline expressions, and **\$\$expression\$\$** for block equations
- **mode** = **selfcontained/standalone/draft** → depending whether the presentation will be given with Internet access or not
 - * **standalone** = all the JavaScript libraries will be save locally so that the presentation can be executed without Internet access
 - * **selfcontained** = load all JavaScript library at time of presentation
- **logo** = displays a logo in title slide
- **url** = specify path to assets/other folders that are used in the presentation
 - * **Note:** *../ signifies the parent directory*

- **example**

```

---
title      : Slidify
subtitle   : Data meets presentation
author     : Jeffrey Leek, Assistant Professor of Biostatistics
job        : Johns Hopkins Bloomberg School of Public Health
logo       : bloomberg_shield.png
framework  : io2012          # {io2012, html5slides, shower, dzslides, ...}
highlighter : highlight.js   # {highlight.js, prettify, highlight}
hitheme    : tomorrow       #
url:
  lib: ../../libraries
  assets: ../../assets
widgets    : [mathjax]       # {mathjax, quiz, bootstrap}
mode       : selfcontained   # {standalone, draft}
---

```

Slides

- **##** = signifies the title of the slide → equivalent of **h1** element in HTML
- **---** = marks the end of a slide
- **.class #id** = assigns **class** and **id** attributes (CSS) to the slide and can be used to customize the style of the page
- **Note:** *make sure to leave space between each component of the slidify document (title, code, text, etc) to avoid errors*
- advanced HTML can be added directly to the **index.Rmd** file and most of the time it should function correctly
- interactive element (quiz questions, rCharts, shiny apps) can be embedded into slidify documents ([demos](#))
 - quiz elements
 - * **##** = signifies title of questions
 - * the question can be type in plain text format

- * the multiple choice options are listed by number (1. a, 2. b, etc.)
 - wrap the correct answer in underscores (2. b)
- * `*** .hint` = denotes the hint that will be displayed when the user clicks on *Show Hint* button
- * `*** .explanation` = denotes the explanation that will be displayed when the user clicks on *Show Answer* button
- * a page like the one below will be generated when processed with slidify

```
## Question 1
```

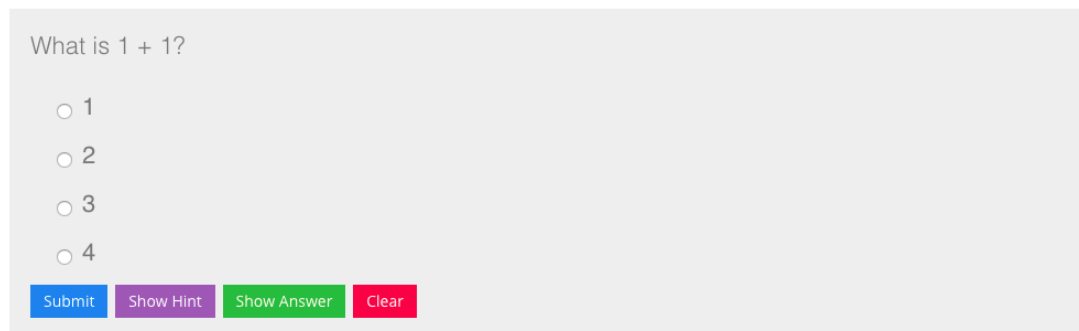
```
What is 1 + 1?
```

- ```
1. 1
2. 2
3. 3
4. 4
```

```
*** .hint
This is a hint
```

```
*** .explanation
This is an explanation
```

## Question 1



What is 1 + 1?

☐ 1

☒ 2

☐ 3

☐ 4

Submit Show Hint Show Answer Clear

- `knit HTML` button can be used to generate previews for the presentation as well

## Publishing

- first, you will need to create a new repository on GitHub
- `publish_github("user", "repo")` can be used to publish the slidify document on to your on-line repo

## RStudio Presentation

- presentation authoring tool within the RStudio IDE ([tutorial](#))
- output = html5 presentation
- .Rpres file -> converted to .md file -> .html file
- uses R Markdown format as slidify/knitr
  - **mathjax** JS library is loaded by default
- RStudio format/runs the code when the document is saved

## Creating Presentation

- file -> New File -> R Presentation (**alt + f + p**)
- **class:** `classname` = specify slide-specific control from CSS
- **css:** `file.css` = can be used to import an external CSS file
  - alternatively, a css file that has the same name as the presentation will be automatically loaded
- knowledge of CSS/HTML/JavaScript useful to customize presentation more granularly
  - **Note:** *though the end HTML file can be edited directly, it should be used as a last resort as it defeats the purpose of reproducible presentations*
- clicking on **Preview** button brings up **Presentation** viewer in RStudio
  - *navigation controls* (left and right arrows) are located in right bottom corner
  - the *Notepad* icon on the menu bar above displays the section of code that corresponds with the current slide in the main window
  - the *More* button has four options
    - \* “Clear Knitr Cache” = clears cache for the generated presentation previews
    - \* “View in Browser” = creates temporary HTML file and opens in default web browser (does not create a local file)
    - \* “Save as Web Page” = creates a copy of the presentation as a web page
    - \* “Publish to RPub” = publishes presentation on RPub
  - the *Refresh* button refreshes the page
  - the *Zoom* button opens a new window to display the presentation
- **transitions between slides**
  - just after the beginning of each slide, the **transition** property (similar to YAML) can be specified to control the transition between the previous and current slides
  - **transition:** `linear` = creates 2D linear transition (html5) between slides
  - **transition:** `rotate` = creates 3D rotating transition (html5) between slides
  - more transition options are found [here](#)
- **hierarchical organization**
  - attribute **type** can be added to specify the appearance of the slide (“slide type”)
  - **type:** `section` and **type:** `sub-section` = distinct background and font colors, slightly larger heading text, appear at a different indent level within the slide navigation menu
  - **type:** `prompt` and **type:** `alert` = distinct background color to communicate to viewers that the slide has different intent
- **columns**
  - simply place **\*\*\*** in between two sections of content on a slide to separate it into two columns
  - **left:** `70%` can be used to specify the proportions of each column

- right: 30% works similarly
- **change slide font ([guide](#))**
  - `font-family: fontname` = changes the font of slide (specified in the same way as HTML)
  - `font-import: http://fonts.googleapis.com/css?family=Risque` = imports font
    - \* ***Note:** fonts must be present on the system for presentation (or have Internet), or default fonts will be used*
  - ***Note:** CSS selectors for class and IDs must be preceded by `.reveal` to work (`.reveal section del` applies to any text enclosed by `~~text~~`)*

## Slidify vs RStudio Presenter

- **Slidify**
  - flexible control from the `.Rmd` file
  - under constant development
  - large user base, more likely to get answer on *StackOverflow*
  - lots of styles and options by default
  - steeper learning curve
  - more command-line oriented
- **R Studio Presenter**
  - embedded in R Studio
  - more GUI oriented
  - very easy to get started
  - smaller set of easy styles and options
  - default styles look nice
  - as flexible as Slidify with CSS/HTML knowledge

## R Package

- R packages = collection of functions/data objects, extends base functionality of R
  - organized to provide consistency
  - written by people all over the world
- primarily available from CRAN and Bioconductor
  - installed with `install.packages()`
- also available on GitHub/BitBucket/etc
  - installed using `devtools::install_github()`
- **documentation/vignettes** = forces author to provide detailed explanation for the arguments/results of their functions and objects
  - allows for well defined Application Programming Interface (API) to tell users what and how to use the functions
  - much of the implementation details can be hidden from the user so that updates can be made without interfering with use cases
- if package is available on CRAN then it must hold standards of reliability and robustness
- fairly easily maintained with proper documentation
- **package development process**
  - write code in R script
  - desire to make code available to others
  - incorporate R script file into R package structure
  - write documentation for user functions
  - include examples/demos/datasets/tutorials
  - package the contents
  - submit to CRAN/Bioconductor
  - push source code repository to GitHub/other code sharing site
    - \* people find problems with code and expect the author to fix it
    - \* alternatively, people might fix the problem and show the author the changes
  - incorporate changes and release a new version

## R Package Components

- **directory** with name of R package = created as first step
- **DESCRIPTION file** = metadata/information about package
- **R code** = code should be in `R/` sub-directory
- **Documentation** = file should be in `man/` sub-directory
- **NAMESPACE** = optional but common and best practice
- full requirements documented in [Writing R Extensions](#)

## DESCRIPTION file

- **Package** = name of package (e.g. `library(name)` to load the package)
- **Title** = full name of package
- **description** = longer description of package in one or two sentences
- **Version** = version number (usually M.m-p format, “majorNumber.minorNumber-patchLevel”)
- **Author** = Name of the original author(s)
- **Maintainer** = name + email of person (maintainer) who fixes problems
- **License** = license for the source code, describes the term that the source code is released under
  - common licenses include GNU/BSD/MIT
  - typically a standard open source license is used
- optional fields
  - **Depends** = R packages that your package depends on
  - **Suggests** = optional R packages that users may want to have installed
  - **Date** = release date in YYYY-MM-DD format
  - **URL** = package home page/link to repository
  - **Other** = fields can be added (generally ignored by R)
- **example: gpclib**
  - **Package:** gpclib
  - **Title:** General Polygon Clipping Library for R
  - **Description:** General polygon clipping routines for R based on Alan Murta’s C library
  - **Version:** 1.5-5
  - **Author:** Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu) with contributions from Duncan Murdoch and Barry Rowlingson; GPC library by Alan Murta
  - **Maintainer:** Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu)
  - **License:** file LICENSE
  - **Depends:** R ( $\geq 2.14.0$ ), methods
  - **Imports:** graphics
  - **Date:** 2013-04-01
  - **URL:** <http://www.cs.man.ac.uk/~toby/gpc/>, <http://github.com/rdpeng/gpclib>

## R Code

- copy R code to R/ directory
- can be any number of files
- separate out files to logical groups (read/fit models)
- all code should be included here and not anywhere else in the package

## NAMESPACE file

- effectively an API for the package
- indicates which functions are *exported* → public functions that users have access to and can use
  - functions not exported cannot be called directly by the user
  - hides the implementation details from the users (clean package interface)
- lists all dependencies on other packages/indicate what functions you *imported* from other packages
  - allows for your package to use other packages without making them visible to the user
  - importing a function loads the package but *does not* attach it to the search list



- **key directives**

- `export("\<function>")` = export a function
- `import("\<package>")` = import a package
- `importFrom("\<package>", "\<function>")` = import specific function from a package
- `exportClasses("\<class>")` = indicate the new types of S4 (4<sup>th</sup> version of *S*) classes created with the package (objects of the specified class can be created)
- `exportMethods("\<generic>")` = methods that can operate on the new class objects
- **Note:** *though they look like R functions, the above directives are not functions that users can use freely*

- **example**

```
read.polyfile/write.polyfile are functions available to user
export("read.polyfile", "write.polyfile")
import plot function from graphics package
importFrom(graphics, plot)
gpc.poly/gpc.poly.nohole classes can be created by the user
exportClasses("gpc.poly", "gpc.poly.nohole")
the listed methods can be applied to the gpc.poly/gpc.poly.nohole classes
exportMethods("show", "get.bbox", "plot", "intersect", "union", "setdiff",
 "[", "append.poly", "scale.poly", "area.poly", "get.pts",
 "coerce", "tristrip", "triangulate")
```

## Documentation

- documentation files (.Rd) should be placed in the `man/` sub-directory
- written in specific markup language
- required for every exported function available to the user (serves to limit the number of exported functions)
- concepts/package/datasets overview can also be documented
- **components**
  - `\name{}` = name of function
  - `\alias{}` = anything listed as alias will bring up the help file (?line is the same as ?residuals.tukeyline)
    - \* multiple aliases possible
  - `\title{}` = full title of the function
  - `\description{}` = full description of the purpose of function
  - `\usage{}` = format/syntax of function
  - `\arguments{}` = explanation of the arguments in the syntax of function
  - `\details{}` = notes/details about limitation/features of the function
  - `\value{}` = specifies what object is returned
  - `\reference{}` = references for the function (paper/book from which the method is created)
- **example: line function**

```

\name{line}
\alias{line}
\alias{residuals.tukeyline}
\title{Robust Line Fitting}
\description{
 Fit a line robustly as recommended in \emph{Exploratory Data Analysis}.
}
\usage{
\code{line}(x, y)
}
\arguments{
 \item{x, y}{the arguments can be any way of specifying x-y pairs. See
 \code{\link{xy.coords}}.}
}
\details{
 Cases with missing values are omitted.

 Long vectors are not supported.
}
\value{
 An object of class \code{"tukeyline"}.

 Methods are available for the generic functions \code{coef},
 \code{residuals}, \code{fitted}, and \code{print}.
}
\references{
 Tukey, J. W. (1977).
 \emph{Exploratory Data Analysis},
 Reading Massachusetts: Addison-Wesley.
}

```

## Building/Checking Package

- **R CMD (Command) Build** = command-line program that creates a package archive file (format = .tar.gz)
- **R CMD Check** = command-line program that runs a battery of tests on the package to ensure structure is consistent/all components are present/export and import are appropriately specified
- R CMD Build/R CMD check can be run from the command-line using terminal/command-shell applications
- alternatively, they can be run from R using the `system()` function
  - `system("R CMD build newpackage")`
  - `system("R CMD check newpackage")`
- the package must pass *all* tests to be put on CRAN
  - documentation exists for all exported function
  - code can be loaded without any major coding problems/errors
  - contains license
  - ensure examples in documentation can be executed
  - check documentation of arguments matches argument in the code
- `package.skeleton()` function in the `utils` package = creates a “skeleton” R package

- automatically creates directory structure (`R/`, `man/`), DESCRIPTION file, NAMESPACE file, documentation files
- if there are any visible/stored functions in the current workspace, their code will be written as R code files in the `R/` directory
- documentation stubs are created in `man/` directory
- the rest of the content can then be modified and added
- alternatively, you can click on the menu on the top right hand corner of RStudio: *Project -> New Project -> New Directory -> R Package* -> fill in package names and details -> automatically generate structure/skeleton of a new R package

## Checklist for Creating Package

- create a new directory with `R/` and `man/` sub-directories (or just use `package.skeleton()`)
- write a DESCRIPTION file
- copy R code into the `R/` sub-directory
- write documentation files in `man/` sub-directory
- write a NAMESPACE file with exports/imports
- build and check

## Example: topten function

- when creating a package, generate skeleton by clicking on *Project -> New Project -> New Directory -> R Package* -> fill in package names and details
- write the code first in a .R script and add documentation directly to the script
  - **Roxygen2** package will be leveraged to extract and format the documentation from R script automatically
- Roxygen2 syntax
  - `#'` = denotes the beginning of documentation
    - \* R will automatically add `#'` on the subsequent lines as you type or complete sections
  - title should be on the first line (relatively concise, a few words)
    - \* press **ENTER** after you are finished and R will automatically insert an empty line and move the cursor to the next section
  - description/summary should begin on the third line (one/two sentences)
    - \* press **ENTER** after you are finished and R will automatically insert an empty line and move the cursor to the next section
  - `@param x definition` = format of the documentation for the arguments
    - \* `x` = argument name (formatted in code format when processed to differentiate from definition)
    - \* `definiton` = explanation of the what x represents
  - `@author` = author of the function
  - `@details` = detailed description of the function and its purpose
  - `@seealso` = links to relevant functions used in creating the current function that may be of interest to the user
  - `@import package function` = imports specific function from specified package
  - `@export` = denotes that this function is exported for public use
  - `@return` = specifies what is returned by the method

```

#' Building a Model with Top Ten Features
#'
#' This function develops a prediction algorithm based on the top 10 features
#' in 'x' that are most predictive of 'y'.
#'
#' @param x a n x p matrix of n observations and p predictors
#' @param y a vector of length n representing the response
#' @return a 'lm' object representing the linear model with the top 10 predictors
#' @author Roger Peng
#' @details
#' This function runs a univariate regression of y on each predictor in x and
#' calculates the p-value indicating the significance of the association. The
#' final set of 10 predictors is the taken from the 10 smallest p-values.
#' @seealso \code{lm}
#' @import stats
#' @export

topten <- function(x, y) {
 p <- ncol(x)
 if(p < 10)
 stop("there are less than 10 predictors")
 pvalues <- numeric(p)
 for(i in seq_len(p)) {
 fit <- lm(y ~ x[, i])
 summ <- summary(fit)
 pvalues[i] <- summ$coefficients[2, 4]
 }
 ord <- order(pvalues)
 x10 <- x[, ord]
 fit <- lm(y ~ x10)
 coef(fit)
}

#' Prediction with Top Ten Features
#'
#' This function takes a set coefficients produced by the \code{topten}
#' function and makes a prediction for each of the values provided in the
#' input 'X' matrix.
#'
#' @param X a n x 10 matrix containing n observations
#' @param b a vector of coefficients obtained from the \code{topten} function
#' @return a numeric vector containing the predicted values

predict10 <- function(X, b) {
 X <- cbind(1, X)
 drop(X %*% b)
}

```

## R Classes and Methods

- represents new data types (i.e. list) and its own structure/functions that R doesn't support yet
- R classes and method -> system for object oriented programming (OOP)
- R is interactive and supports OOP where as a lot of the other common OOP languages (C++, Java, Python, Perl) are generally not interactive
- John Chambers wrote most of the code support classes/methods (documented in *Programming with Data*)
- goal is to allow *user* (leverage system capabilities) to become *programmers* (extend system capabilities)
- OOB structure in R is structured differently than most of the other languages
- **two style of classes and methods**
  - S3 (version three of the *S* language)
    - \* informal, “old-style”, kludgy (functional but not elegant)
  - S4 (version four of the *S* language)
    - \* rigorous standard, more formal, “new-style”
    - \* code for implementing S4 classes and methods found in **methods** package
  - two systems living side by side
    - \* each is independent but S4 are encouraged for new projects

## Object Oriented Programming in R

- **class** = description of some thing/object (new data type/idea)
  - can be defined using **setClass()** function in **methods** package
  - all objects in R have a class, which can be determined through the **class()** function
    - \* **numeric** = number data, can be vectors as well (series of numbers)
    - \* **logical** = TRUE, FALSE, NA
      - **Note:** NA is by default of logical class, however you can have numeric/character NA's as results of operations
    - \* **character** = string of characters
    - \* **lm** = linear model class, output from a linear model
- **object** = instances of a class
  - can be created using **new()**
- **method** = function that operates on certain class of objects
  - also can be thought of as an implementation of a *generic function* for an object of particular class
  - can write new method for existing generic functions or create new generic function and associated methods
  - **getS3method(<genericFunction>, <class>)** = returns code for S3 method for a given class
    - \* some S3 methods can be called directly (i.e. **mean.default**)
    - \* but you should **never** call them, always use the generic
  - **getMethod(<genericFunction>, <signature/class>)** = returns code for S4 method for a given class
    - \* *signature* = character vector indicating class of objects accepted by the method

- \* S4 methods can not be called at all
- **generic function** = R function that dispatches methods to perform a certain task (i.e. `plot`, `mean`, `predict`)
  - performs different calculations depending on context
  - **Note:** *generic functions themselves don't perform any computation; typing the function name by itself (i.e. `plot`) will return the content of the function*
  - S3 and S4 functions look different but are similar conceptually
  - `methods("mean")` = returns methods associated with S3 generic function
  - `showMethods("show")` = returns methods associated with S4 generic function
    - \* **Note:** *show is equivalent of `print`, but generally not called directly as objects are auto-printed*
  - first argument = object of particular class
  - **process:**
    1. generic function checks class of object
    2. if appropriate method for class exists, call that method on object (process complete)
    3. if no method exists for class, search for default method
    4. if default method exists, default method is called on object (process complete)
    5. if no default method exists, error thrown (process complete)
  - for classes like `data.frame` where each column can be of different class, the function uses the methods correspondingly
    - \* plotting `as.ts(x)` and `x` are completely different
      - `as.ts()` = converts object to time series
- **Note:** `?Classes`, `?Methods`, `?setClass`, `?setMethod`, and `?setGeneric` contains very helpful documentation
- **example**

```
S3 method: mean
mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fbde4a7a1c8>
<environment: namespace:base>
```

```
associated methods
methods("mean")
```

```
[1] mean.Date mean.default mean.difftime mean.POSIXct mean.POSIXlt
```

```
code for mean (first 10 lines)
note: no specific function got numeric class, so default is used
head(getS3method("mean", "default"), 10)
```

```
##
1 function (x, trim = 0, na.rm = FALSE, ...)
2 {
3 if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
4 warning("argument is not numeric or logical: returning NA")
```

```
5 return(NA_real_)
6 }
7 if (na.rm)
8 x <- x[!is.na(x)]
9 if (!is.numeric(trim) || length(trim) != 1L)
10 stop("'trim' must be numeric of length one")
```

```
S4 method: show
show
```

```
standardGeneric for "show" defined from package "methods"
##
function (object)
standardGeneric("show")
<bytecode: 0x7fbde3cd5c78>
<environment: 0x7fbde2f6af50>
Methods may be defined for arguments: object
Use showMethods("show") for currently available ones.
(This generic function excludes non-simple inheritance; see ?setIs)
```

```
associated methods
showMethods("show")
```

```
Function: show (package methods)
object="ANY"
object="C++Class"
object="C++Function"
object="C++Object"
object="classGeneratorFunction"
object="classRepresentation"
object="color"
object="Enum"
object="EnumDef"
object="envRefClass"
object="function"
(inherited from: object="ANY")
object="genericFunction"
object="genericFunctionWithTrace"
object="MethodDefinition"
object="MethodDefinitionWithTrace"
object="MethodSelectionReport"
object="MethodWithNext"
object="MethodWithNextWithTrace"
object="Module"
object="namedList"
object="ObjectsWithPackage"
object="oldClass"
object="refClassRepresentation"
object="refMethodDef"
object="refObjectGenerator"
object="signature"
object="sourceEnvironment"
object="standardGeneric"
```

```
(inherited from: object="genericFunction")
object="SymbolicConstant"
object="traceable"
```

## Creating a New Class/Methods

- reason for creating new classes/data type (not necessarily unknown to the world, but just unknown to R)
  - powerful way to extend the functionality of R
  - represent new types of data (e.g. gene expression, space-time, hierarchical, sparse matrices)
  - new concepts/ideas that haven't been thought of yet (e.g. a fitted point process model, mixed-effects model, a sparse matrix)
  - abstract/hide implementation details from the user
- **classes** = define new data types
- **methods** = extend *generic functions* to specify the behavior of generic functions on new classes
- **setClass()** = function to create new class
  - at minimum, name of class needs to be specified
  - *slots* or attributes can also be specified
    - \* a class is effectively a list, so slots are elements of that list
- **setMethod()** = define methods for class
  - @ is used to access the slots/attributes of the class
- **showClass()** = displays definition/information about class
- when drafting new class, new methods for **print**, **show**, **summary**, and **plot** should be written
- **Note:** creating classes are not something to be done on the console and are much better suited for a script
- **example**
  - create **polygon** class with set of (x, y) coordinates with **setClass()**
  - define a new plot function by extending existing **plot** function with **setMethod()**

```
load methods library
library(methods)
create polygon class with x and y coordinates as slots
setClass("polygon", representation(x = "numeric", y = "numeric"))
create plot method for ploygon class (ploygon = signature in this case)
setMethod("plot", "polygon",
 # create function
 function(x, y, ...) {
 # plot the x and y coordinates
 plot(x@x, x@y, type = "n", ...)
 # plots lines between all (x, y) pairs
 # x@x[1] is added at the end because we need
 # to connect the last point of polygon to the first
 xp <- c(x@x, x@x[1])
 yp <- c(x@y, x@y[1])
 lines(xp, yp)
 })
```



```
Creating a generic function for 'plot' from package 'graphics' in the global environment
```

```
[1] "plot"
```

```
print polygon method
showMethods("plot")
```

```
Function: plot (package graphics)
```

```
x="ANY"
```

```
x="color"
```

```
x="polygon"
```

## Yhat ([link](#))

- develop back-ends to algorithms to be hosted on-line for other people to access
- others can create APIs (front-ends) to leverage the algorithm/model
- before uploading, create an account and set up API key

```
Create dataset of PM and O3 for all US taking year 2013 (annual
data from EPA)

This uses data from
http://aqsdr1.epa.gov/aqsweb/aqstmp/airdata/download_files.html

Read in the 2013 Annual Data
d <- read.csv("annual_all_2013.csv", nrow = 68210)
subset data to just variables we are interested in
sub <- subset(d, Parameter.Name %in% c("PM2.5 - Local Conditions", "Ozone")
 & Pollutant.Standard %in% c("Ozone 8-Hour 2008", "PM25 Annual 2006"),
 c(Longitude, Latitude, Parameter.Name, Arithmetic.Mean))
calculate the average pollution for each location
pollavg <- aggregate(sub[, "Arithmetic.Mean"],
 sub[, c("Longitude", "Latitude", "Parameter.Name")],
 mean, na.rm = TRUE)
refactors the Name parameter to drop all other levels
pollavg$Parameter.Name <- factor(pollavg$Parameter.Name, labels = c("ozone", "pm25"))
renaming the last column from "x" (automatically generated) to "level"
names(pollavg)[4] <- "level"

Remove unneeded objects
rm(d, sub)
extract out just the location information for convenience
monitors <- data.matrix(pollavg[, c("Longitude", "Latitude")])
load fields package which allows us to calculate distances on earth
library(fields)
build function to calculate the distances for the given set of coordinates
input = lon (longitude), lat (latitude), radius (radius in miles for finding monitors)
pollutant <- function(df) {
 # extract longitude/latitude
 x <- data.matrix(df[, c("lon", "lat")])
 # extract radius
 r <- df$radius
 # calculate distances between all monitors and input coordinates
 d <- rdist.earth(monitors, x)
 # locations for find which distance is less than the input radius
 use <- lapply(seq_len(ncol(d)), function(i) {
 which(d[, i] < r[i])
 })
 # calculate levels of ozone and pm2.5 at each selected locations
 levels <- sapply(use, function(idx) {
 with(pollavg[idx,], tapply(level, Parameter.Name, mean))
 })
 # convert to data.frame and transpose
 dlevel <- as.data.frame(t(levels))
 # return the input data frame and the calculated levels
 data.frame(df, dlevel)
```

```
}
```

## Deploying the Model

- once the functions are complete, three more functions should be written in order to upload to yhat,
  - `model.require(){} = defines dependencies on other packages`
    - \* if there are no dependencies, this does not need to be defined
  - `model.transform(){} = needed if the data needs to be transformed in anyway before feeding into the model`
  - `model.predict(){} = performs the prediction`
- store the following information as a vector named `yhat.config`
  - `username = "<user@email.com>" = user name for yhat website`
  - `apikey = "<generatedKey>" = unique API key generated when you open an account with yhat`
  - `env="http://sandbox.yhathq.com/" = software environment (always going to be this link)`
- `yhat.deploy("name") = uploads the model to yhat servers with provided credentials under the specified name`
  - returns a data frame with status/model information

```
Send to yhat
library(yhatr)

model.require <- function() {
 library(fields)
}

model.transform <- function(df) {
 df
}

model.predict <- function(df) {
 pollutant(df)
}

yhat.config <- c(
 username="email@gmail.com",
 apikey="90d2a80bb532cabb2387aa51ac4553cc",
 env="http://sandbox.yhathq.com/"
)

yhat.deploy("pollutant")
```

## Accessing the Model

- once uploaded, the model can be accessed directly on the yhat website
  - click on name of the model after logging in or go to ["http://cloud.yhathq.com/model/name"](http://cloud.yhathq.com/model/name) where name is the name you uploaded the model under
  - enter the inputs in JSON format (associative arrays): `{ "variable" : "value" }`
    - \* *example:* `{ "lon" : -76.61, "lat": 39.28, "radius": 50 }`

- click on **Send Data to Model**
- results return in the *Model Response* section
- the model can also be accessed from R directly through the `yhat.predict` function
  - store the data you want to predict on in a data frame with the correct variable names
  - set up the configuration information through `yhat.config` (see above section)
  - `yhat.predict("name", df)` = returns the result by feeding the input data to the model hosted on yhat under your credentials
  - can be applied to multiple rows of data at the same time

```
library(yhatr)
yhat.config <- c(
 username="email@gmail.com",
 apikey="90d2a80bb532cabb2387aa51ac4553cc",
 env="http://sandbox.yhathq.com/"
)
df <- data.frame(lon = c(-76.6167, -118.25), lat = c(39.2833, 34.05),
 radius = 20)
yhat.predict("pollutant", df)
```

- the model can also directly from command line interfaces (CLI) such as cmd on Windows and terminal on Mac

```
curl -X POST -H "Content-Type: application/json" \
 --user email@gmail.com:90d2a80bb532cabb2387aa51ac4553cc \
 --data '{ "lon" : -76.61, "lat": 39.28, "radius": 50 }' \
 http://cloud.yhathq.com/rdpeng@gmail.com/models/pollutant/
```

- *additional example*

```
load library
library(yhatr)
yhat functions
model.require <- function() {}
model.transform <- function(df) {
 transform(df, Wind = as.numeric(as.character(Wind)),
 Temp = as.integer(as.character(Temp)))
}
model.predict <- function(df) {
 result <- data.frame(Ozone = predict(fit, newdata = df))
 cl <- data.frame(clWind = class(df$Wind), clTemp = class(df$Temp))
 data.frame(result, Temp = as.character(df$Temp),
 Wind = as.character(df$Wind), cl)
}
model
fit <- lm(Ozone ~ Wind + Temp, data = airquality)
configuration
yhat.config <- c(
 username="email@gmail.com",
 apikey="90d2a80bb532cabb2387aa51ac4553cc",
 env="http://sandbox.yhathq.com/"
)
```

```
deploy to yhat
yhat.deploy("ozone")
predict using uploaded model
yhat.predict("ozone", data.frame(Wind = 9.7, Temp = 67))
```