



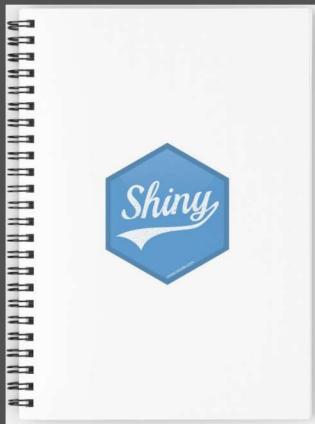
Formation R Shiny

Julien Renault
Octobre 2019

Plan de la formation

Introduction: Pourquoi utiliser Shiny?

1. Les concepts fondamentaux
2. La construction de l'UI
3. La construction du server
4. Déploiement des applications
5. L'écosystème Shiny
6. Exercice final



Pourquoi utiliser Shiny?

Exemples d'applications

Shiny peut servir dans de nombreux domaines où l'interactivité est importante.

Voici quelques exemples:

Exemple 1 : Exploration de données (<https://vnijs.shinyapps.io/radiant/>)

Exemple 2 : Dashboard (analyse des ventes) (<https://mdancho84.shinyapps.io/shiny-app/>)

Exemple 3 : Données spatiales (<https://poissonconsulting.shinyapps.io/rtide/>)

Exemple 4 : Utilisation de Google Cloud Vision (<https://flovv.shinyapps.io/gVision-shiny/>)

Exemple 5 : Analyse de textes (<http://shiny.rstudio.com/gallery/word-cloud.html>)

Exécution de R au sein d'une présentation

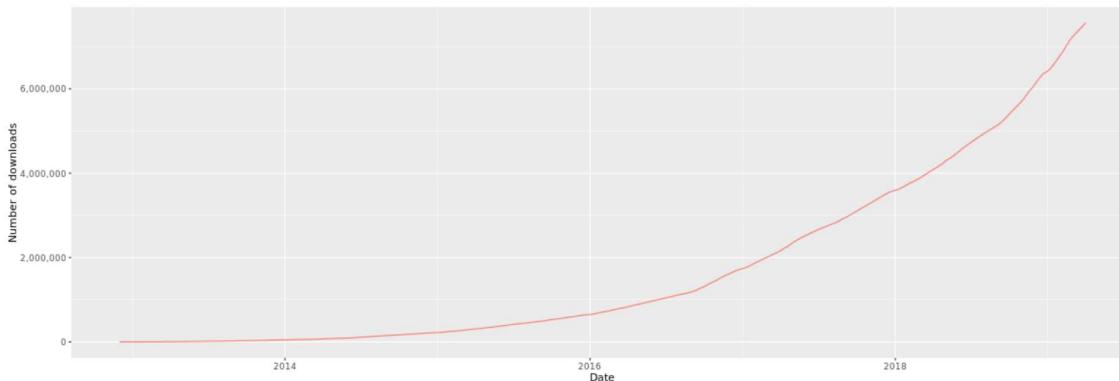
Cette présentation est rédigée en R Markdown en utilisant `ioslides` et le moteur `shiny`.

On peut donc exécuter du code R et afficher l'output:

Taper l'expression ci-dessous

Popularité du package

On peut obtenir les statistiques de téléchargement des packages R, dont Shiny, avec cette application en ligne (<https://shinyus.ipub.com/cranview/>)



La version la plus récente (1.2.0.9001) date du 12 avril 2019 (<https://cran.r-project.org/src/contrib/Archive/shiny/>), mais la première publication de RStudio concernant Shiny date de fin 2012

Le site de RStudio (<http://shiny.rstudio.com/>) est très riche en références et tutoriels pour se perfectionner.

Raison 1 : Simplicité

R Shiny = R + interactivité + web facilité

Shiny est relativement facile à apprendre et à utiliser, même pour des personnes qui ne connaissent rien aux trois langages principaux du web qui sont:

- le `html` qui détermine le contenu et la structure
- le `css` qui contrôle l'aspect (couleurs, polices de caractères, etc.)
- le `javascript` qui permet des animations

Une connaissance de R est suffisante pour accéder à une interface appelant des fonctionnalités, elles-mêmes originellement codées avec ces langages.

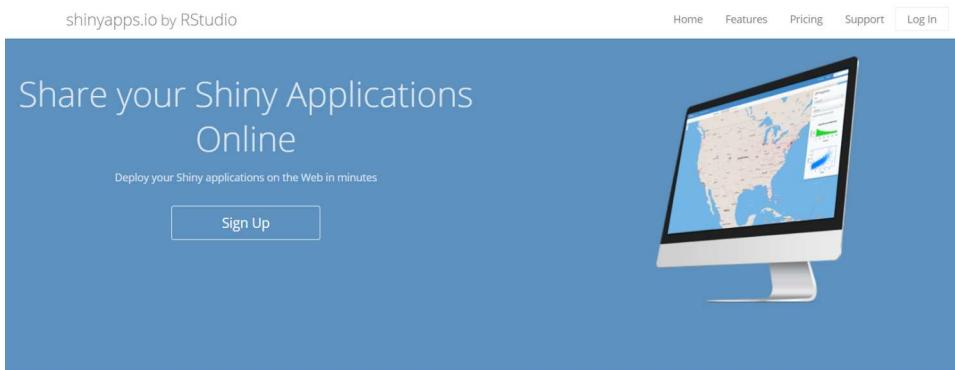


Raison 2 : Rapidité de développement

Le développement d'une application Shiny est souvent rapide à opérer. Son utilisation permet donc d'accélérer la mise sur le marché de données, de dataviz ou de modèles de machine learning.

L'installation de R, RStudio et de quelques librairies est suffisante pour faire une application de A à Z.

Une fois l'application développée, il existe plusieurs solutions au sein même de RStudio pour la déployer en ligne.



Raison 3 : Excellente solution de Dataviz

Shiny est une des rares solutions sur le marché permettant avec des outils **gratuits** de faire de la dataviz de qualité. A peu près n'importe quel graphique R peut être rendu interactif avec Shiny.

Shiny a de plus le gros avantage sur beaucoup solutions (souvent trop statiques) de pouvoir ajouter de l'interactivité aux visualisations.

Les applications comprenant une dimension géospatiale sont particulièrement efficaces sous Shiny.

Il suffit d'explorer un site comme [Data-to-Viz](https://www.data-to-viz.com/) (<https://www.data-to-viz.com/>) pour apprécier la qualité de ce qu'il est possible de faire.



Raison 4 : le language R

Shiny s'adresse aux personnes connaissant déjà les bases de R, qui est maintenant un langage largement connu et répandu dans de nombreuses entreprises.

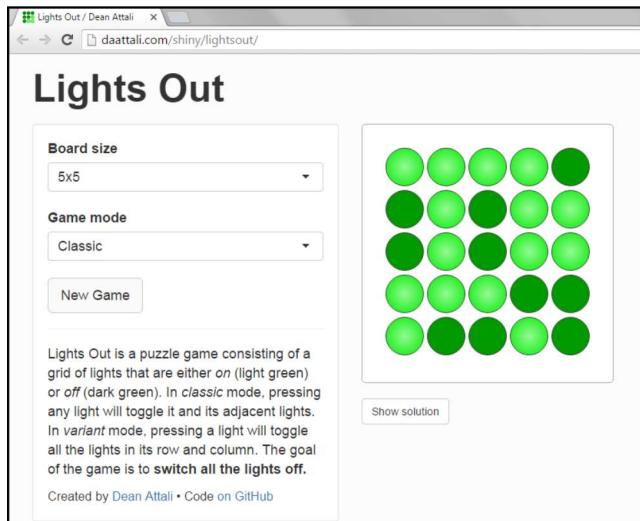
Malgré la concurrence de plus en plus grande avec Python, il reste très compétitif pour le prototypage et l'analyse exploratoire, mais il peut aussi aller bien au delà des ces besoins. Des centaines de compagnies l'utilisent au quotidien :



Raison 5 : Potentiel très large

Les possibilités offertes par Shiny sont immenses, comme nous l'avons vu dans les exemples introductifs.

Certains se sont même amusé à l'utiliser pour développer des jeux (<https://daattali.com/shiny/lightsout/>) sous Shiny.





Les concepts fondamentaux

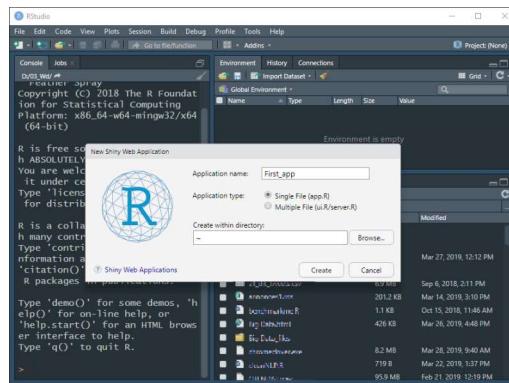
Prérequis et configuration

Pour commencer à coder en Shiny, vous aurez simplement besoin :

A. D'installer R et le package R Shiny

```
install.packages("shiny")
```

B. D'installer RStudio



Qu'est qu'une application Shiny?

C'est une page web (UI = User Interface) connectée à un serveur (qui peut être un simple laptop) où tourne R.

Les utilisateurs peuvent manipuler l'UI, ce qui provoque des changements dans le serveur afin de mettre à jour ce qui est affiché (en exécutant du code R).

Ceci est l'idée de base de l'expression réactive sur lequel Shiny repose.

```
library(shiny)

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

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

Pour utiliser des termes de développement web, on peut dire qu'un développeur Shiny est **fullstack**, car il maîtrise:

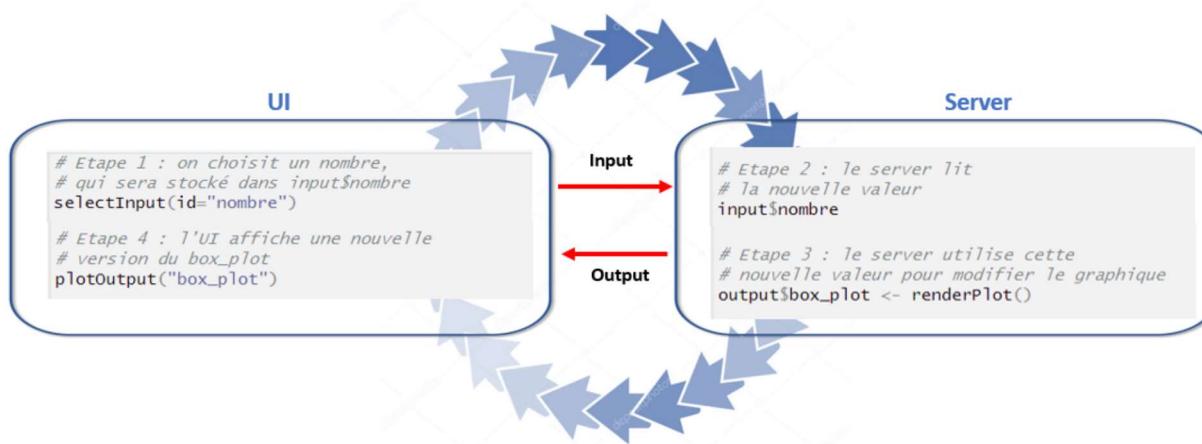
le **backend**, qu'on appelle ici le **server**, qui est responsable de la logique de l'application,
le **frontend**, qu'on appelle ici l'**UI**, qui permet de définir l'aspect de l'application.

Un framework “Server-based”

Contrairement à d'autres frameworks comme D3.js où la machine client doit faire tous les calculs pour la visualisation, Shiny a l'avantage de ne pas reposer sur la puissance de la machine client car les calculs sont faits sur un serveur.

Ceci s'avère très pratique lorsque l'application est hébergée en ligne.

Le cycle ui-serveur peut être représenté simplement :



Qu'est-ce que la réactivité?

Prenons un script R très simple:

```
a <- 100  
b <- a+1  
  
a <- 200
```

Si on le lance, on aura `a == 200` et `b == 101`.

Dans un script R Shiny, ce sera différent : `a == 200` et `b == 201` du fait de la réactivité.

Autrement dit, la valeur de `b` est automatiquement mise à jour lorsque `a` change, sans qu'il soit nécessaire de redonner l'instruction `b <- a+1`.

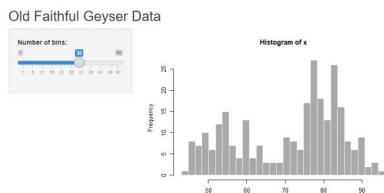
Typiquement la valeur de `a` sera un **input** de l'application et `b` un **output**.

Le template RStudio

Une façon simple de créer une application est de la créer au moyen du menu File / New File de RStudio:



Ceci donne le code pour exécuter l'application ci-dessous, dont on peut ensuite modifier le code :

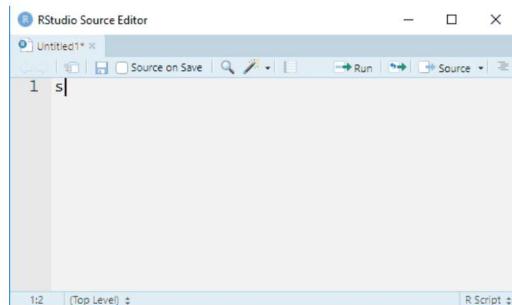


Remarque : RStudio affiche un bouton "Run App" au-dessus du script à partir du moment où ce script se termine par `shinyApp(ui, server)`.

Exemple : `template_demo.R`

Template à l'aide d'un snippet RStudio

Une autre façon de créer un squelette d'application shiny est d'utiliser un **snippet** dans l'éditeur de script:



Ceci va automatiquement générer le code suivant:

```
library(shiny)
ui <- fluidPage(
  )
server <- function(input, output, session) {
  }
shinyApp(ui, server)
```

Options d'écriture et de lancement d'une application

Dans la plupart des exercices et démos, nous allons voir des applications **single file** dans le sens où toute l'application est écrite dans un seul script R.

Il est également possible d'écrire un script `ui.R` ainsi que `server.R` et de lancer l'application en pointant sur le dossier contenant ces 2 scripts

```
shinyAppDir("nom_du_dossier")
```

Par ailleurs, lorsqu'on lance une application, on peut utiliser le mode "showcase" qui:

- affichera le code de l'application
- surlignera la partie du code influencée par un changement d'input
- fera apparaître du texte explicatif contenu dans un fichier markdown `Readme.md`

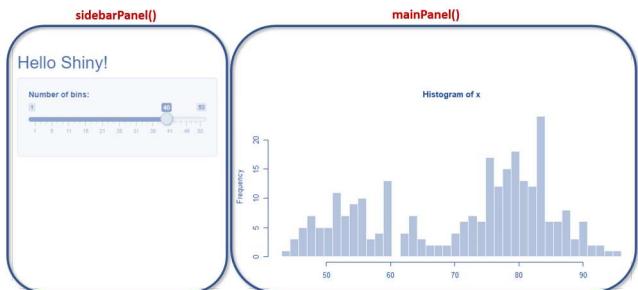
```
runApp("D:/01_Soft_Computing/Data_Science/Formation_Shiny/live/02_Demos", display.mode = "showcase")
```

UI basique

Si l'on ne part pas d'un template, la première chose à faire est de choisir l'agencement (**layout** en anglais) des différentes parties de l'application.

La façon la plus simple d'organiser l'interface est de placer un **sidebarPanel** (pour les inputs) à gauche du **mainPanel** (pour les outputs)

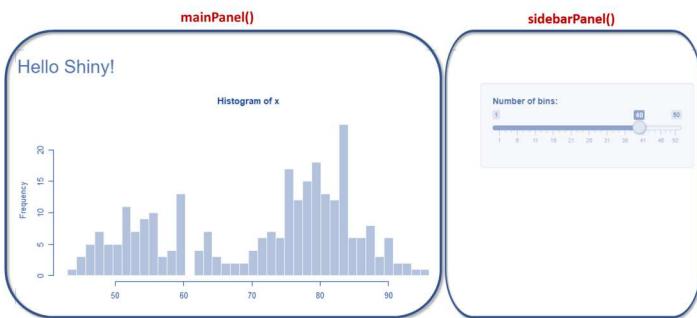
```
sidebarLayout(  
  sidebarPanel(  
    # Inputs  
  ),  
  mainPanel(  
    # Outputs  
  ))
```



Emplacement du sidebarPanel

Il est également possible de placer le sidebarPanel à droite en écrivant

```
sidebarLayout(  
  position = "right",  
  sidebarPanel(  
    # Inputs  
  ),  
  mainPanel(  
    # Outputs  
  ))
```

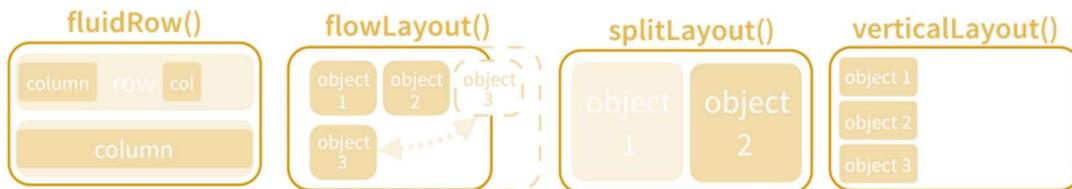




La construction de l'UI

Les agencements (layout / panel)

Selon la complexité de l'application, on peut ajouter d'autres types de panels que le sidebar et le main.



Les panels permettent de réunir différents éléments, et il est possible de les emboîter.

- `conditionalPanel()` permet d'afficher un objet selon la valeur de l'input
- `tabsetPanel()` permet d'afficher des onglets pour subdiviser le `mainPanel` en plusieurs `tabPanel`
- `absolutePanel()` permet de positionner les éléments du `mainPanel` de manière fixe
- `verticalLayout()` permet d'afficher les éléments de l'UI verticalement
- `splitLayout()` permet d'afficher les éléments de l'UI horizontalement, en colonnes égales
- `flowLayout()` permet d'aligner les éléments de gauche à droite

La fonction conditionalPanel()

Cette fonction permet de faire apparaître (resp. cacher) des parties de l'UI en fonction de sélections.

```
ui <- fluidPage(
  column(4,
    wellPanel(
      sliderInput(
        "n", "Number of points:",
        min = 10, max = 200, value = 50, step = 10))),
  column(5,
    conditionalPanel("input.n >= 50",
      plotOutput("scatterPlot", height = 300))
  )))

```

Remarque : cette fonction fait appel à du Javascript, ce qui explique que l'appel de fonction soit avec `input.n` plutôt que `input$n`.

Pour cette raison, il est préférable de ne pas mettre de points dans le nom de la variable (ici `n`).

Exemple : `conditionalPanel_demo.R`

La fonction `tabPanel()` avec `navbarPage()`

La fonction `navbarPage()` crée un menu dans lequel on peut encapsuler des sous-menus avec `tabPanel()`.

```
ui <- fluidPage(
  navbarPage("Cars dataset", tabPanel("Graphiques",
    sidebarLayout(sidebarPanel(radioButtons(
      "plotType", "Plot type", c("Scatter" = "p", "Line" = "l"))),
    mainPanel(plotOutput("plot")))),
  tabPanel("Stats descriptives",
    verbatimTextOutput("summary")),
))
```

Remarque : pour ajouter des icônes dans les menus, il suffit d'indiquer le nom d'une icône disponible dans la librairie [Font Awesome](https://fontawesome.com/icons?d=listing&m=free) (<https://fontawesome.com/icons?d=listing&m=free>).

```
tabPanel("Tableau",
  DT::dataTableOutput("table"),
  icon = icon("table"))
```

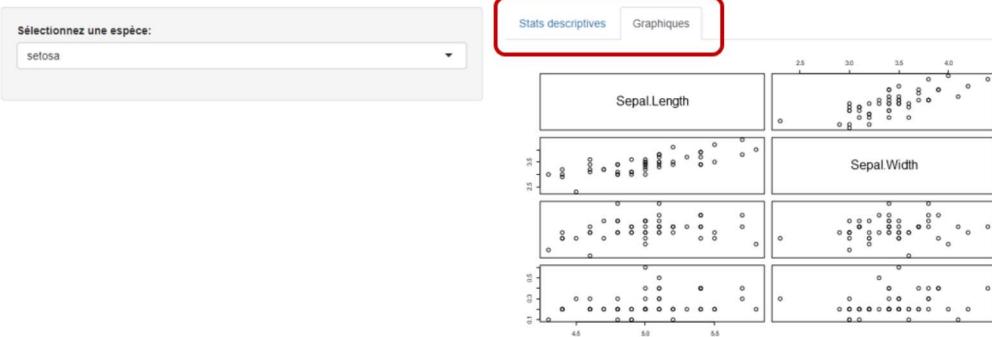
Exemple : `navbar_demo.R`

La fonction tabsetPanel()

Cette fonction permet de créer des onglets en imbriquant par exemple 2 tabPanel()

```
tabsetPanel(  
  tabPanel("Stats descriptives", tableOutput("table")),  
  tabPanel("Graphiques", plotOutput("plot")))
```

Exemple de TabsetPanel

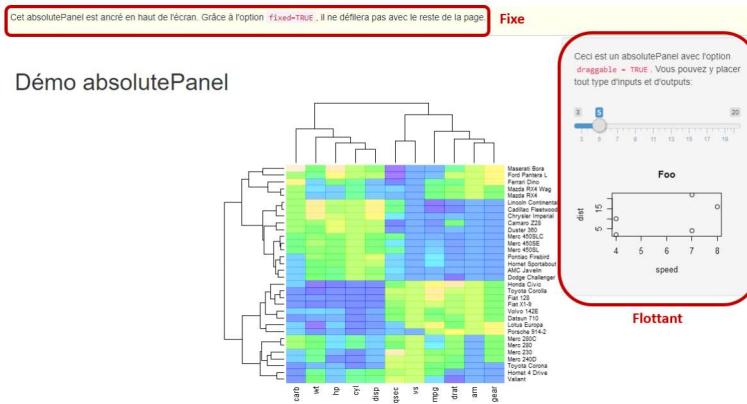


Exemple : tabsetPanel_demo.R

La fonction absolutePanel()

Cette fonction permet de créer des panels de tailles et positions fixes. On peut par exemple:

- positionner un panel en haut de la page avec l'option `fixed = TRUE`
- faire en sorte que ce panneau soit flottant avec l'option `draggable = TRUE`

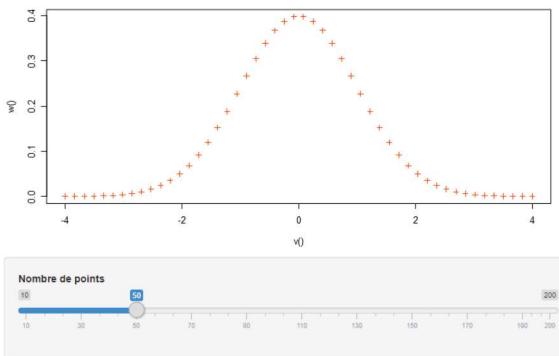


Exemple : `absolutePanel_demo.R`

La fonction verticalLayout()

Comme son nom l'indique, cette fonction permet de positionner les éléments de façon verticale.

```
ui <- fluidPage(  
  verticalLayout(  
    titlePanel("Exemple de vertical layout"),  
    plotOutput("plot1"))
```

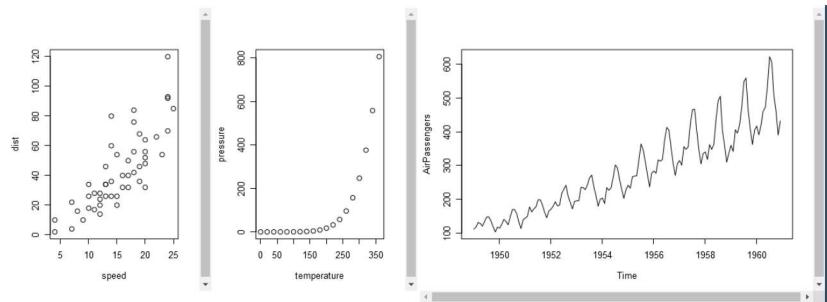


Exemple : `verticalLayout_demo.R`

La fonction splitLayout()

Cette fonction permet par exemple d'agencer les éléments selon des ratios prédéfinis :

```
ui <- fluidPage( splitLayout(  
  cellWidths = c("25%", "25%", "50%"),  
  plotOutput("plot1"),  
  plotOutput("plot2"),  
  plotOutput("plot3"))
```



Exemple : [splitlayout_demo.R](#)

La fonction `flowLayout()`

Cette fonction permet par exemple d'agencer les éléments automatiquement les uns après les autres (de gauche à droite, et de haut en bas) :

```
ui <- fluidPage( flowLayout(  
  numericInput("rows", "Combien de lignes?", 5),  
  selectInput("letter", "Quelle lettre?", LETTERS),  
  sliderInput("value", "Quelle valeur?", 0, 100, 50)  
)
```

The screenshot shows a Shiny application interface. It consists of three input fields arranged horizontally. From left to right: 1) A numeric input field labeled "How many rows?" with the value "5". 2) A select input field labeled "Which letter?" with the value "A". 3) A slider input field labeled "What value?" with the value set to 50, indicated by a blue tick mark and the number "50" inside the slider's track.

Exemple : `flowlayout_demo.R`

Remarque sur la nature sous-jacente de l'UI

Nous avons dit précédemment que l'UI était une page web. Pour s'en convaincre, il suffit de créer un UI...

```
library(shiny)
ui <- fluidPage(
  titlePanel("Ceci est un titre"),
  sidebarLayout(
    sidebarPanel("emplacement des inputs"),
    mainPanel("emplacement des outputs")
))
```

... et ensuite d'afficher l'objet:

```
print(ui)

## <div class="container-fluid">
##   <h2>Ceci est un titre</h2>
##   <div class="row">
##     <div class="col-sm-4">
##       <form class="well">emplacement des inputs</form>
##     </div>
##     <div class="col-sm-8">emplacement des outputs</div>
##   </div>
## </div>
```

Les tags HTML

Pour aller un peu plus loin sur la façon dont Shiny gère le HTML, on peut lister quelques tags disponibles (parmi une liste de 110) :

```
taglist <- shiny::tags  
names(taglist)[1:10]  
  
## [1] "a"      "abbr"    "address" "area"    "article" "aside"   "audio"  
## [8] "b"      "base"    "bdi"
```

Exemple d'utilisation:

```
tags$h1("Titre 1")
```

Titre 1

```
tags$h2("Titre 2")
```

Titre 2

Application des tags HTML

On peut modifier la présentation du texte de l'UI à l'aide des tags:

```
ui = fluidPage(titlePanel(  
  div("Des goûts et des couleurs:", style = "color: green")),  
  sidebarLayout(sidebarPanel(  
    ("Mes plats favoris sont:"),  
    br(),  
    h4(strong("la pizza"), "de chez", em("Papa Raffaele")),  
    h3(strong("les sushis"), "de chez", code("Jomon")),  
    div(strong("les tagines"), "de la Menara", style = "color: red")  
  ),  
  mainPanel()))  
server = function(input, output) {}  
shinyApp(ui, server)
```

Des goûts et des couleurs:

Mes plats favoris sont:
la pizza de chez *Papa Raffaele*
les sushis de chez *Jomon*
les tagines de la Menara

Ajout d'inputs dans l'UI

Les inputs de l'application peuvent être par exemple:

- une valeur numérique avec `numericInput()`,
- du texte avec `textInput()`,
- une date avec `dateInput()`,

Afin de régler la façon dont l'utilisateur saisit le(s) input(s), il existe ce qu'on appelle des **widgets**.

Toutes les fonctions d'input prennent au moins 2 **arguments** :

- `inputId` qui est le nom sous lequel Shiny va reconnaître l'input.
Il doit par conséquent être unique, avec pour valeur `input$<inputId>`
- `label` qui va servir de descriptif dans les widgets

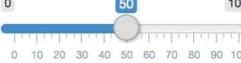
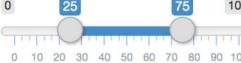


Chacune de ces fonctions peut avoir d'autres arguments spécifiques.

Les widgets de contrôle

Gallerie (<http://shiny.rstudio.com/gallery/widget-gallery.html>)

Basic widgets

Buttons <input type="button" value="Action"/> <input type="button" value="Submit"/>	Single checkbox <input checked="" type="checkbox"/> Choice A	Checkbox group <input checked="" type="checkbox"/> Choice 1 <input type="checkbox"/> Choice 2 <input type="checkbox"/> Choice 3	Date input 2014-01-01
Date range 2017-06-21 to 2017-06-21	File input Browse... No file selected	Help text Note: help text isn't a true widget, but it provides an easy way to add text to accompany other widgets.	Numeric input 1
Radio buttons <input checked="" type="radio"/> Choice 1 <input type="radio"/> Choice 2 <input type="radio"/> Choice 3	Select box Choice 1	Sliders  	Text input Enter text...

Widgets de contrôle

Input numérique

L'input numérique permet de sélectionner une valeur numérique.

```
numericInput(inputId, label,  
            min = NA, max = NA, step = NA)
```

Numeric input

A numeric input control consisting of a text field containing the number '1' and two small arrows (up and down) on the right side, used for incrementing or decrementing the value.

Exemple : numeric_input.R

Widgets de contrôle

Boutons radio

Les **boutons radio** permettent de sélectionner une valeur numérique (unique) au sein d'une liste de choix.

```
ui <- fluidPage(  
  radioButtons("radio",  
    label = h3("Radio buttons"),  
    choices = list(  
      "Choice 1" = 1,  
      "Choice 2" = 2,  
      "Choice 3" = 3), selected = 1)
```

Exemple : radio_buttons.R

Radio buttons

- Choice 1
- Choice 2
- Choice 3

Widgets de contrôle

Slider

Le **slider** permet de sélectionner une valeur numérique dans un intervalle déterminé en utilisant un curseur.

Il existe de nombreuses options permettant de définir l'intervalle, le format, l'animation, etc.



```
sliderInput("integer", "Integer:",
           min = 0, max = 1000,
           value = 500),

sliderInput("decimal", "Decimal:",
           min = 0, max = 1,
           value = 0.5, step = 0.1),

sliderInput("range", "Range:",
           min = 1, max = 1000,
           value = c(200,500)),

sliderInput("format", "Custom Format:",
           min = 0, max = 10000,
           value = 0, step = 2500,
           pre = "$", sep = ".",
           animate = TRUE),

sliderInput("animation", "Looping Animation:",
           min = 1, max = 2000,
           value = 1, step = 10,
           animate = TRUE,
           animationOptions(interval = 300, loop = TRUE))
```

Exemple : sliders_input.R

Widgets de contrôle

Checkbox

La **checkbox** permet de sélectionner une (ou plusieurs) valeur(s) en cochant des cases au choix.

```
checkboxGroupInput(inputId, label,  
                   choices = NULL, selected = NULL,  
                   inline = FALSE, width = NULL,  
                   choiceNames = NULL,  
                   choiceValues = NULL)
```

Checkbox group

- Choice 1
- Choice 2
- Choice 3

Exemple : checkbox_input.R

Widgets de contrôle

Input select box

L'input select box permet de sélectionner une valeur en sélectionnant dans un menu déroulant.

```
selectInput(inputId, label, choices,  
           selected = NULL, multiple = FALSE,  
           selectize = TRUE,  
           width = NULL, size = NULL)
```



Exemple : `select_input.R`

Widgets de contrôle

Input date

L'input date permet de sélectionner une date en sélectionnant dans le calendrier.

```
dateInput(inputId, label,  
         value = NULL, min = NULL, max = NULL,  
         format = "yyyy-mm-dd", startview = "month",  
         weekstart = 0, language = "en",  
         width = NULL, autoclose = TRUE,  
         datesdisabled = NULL,  
         daysofweekdisabled = NULL)
```



Exemple : date_input.R

Widgets de contrôle

Input texte

L'input texte permet de saisir du texte.

```
selectInput(inputId, label, choices,  
           selected = NULL, multiple = FALSE,  
           selectize = TRUE,  
           width = NULL, size = NULL)
```



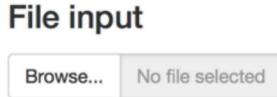
Exemple : text_input.R

Widgets de contrôle

Input file

L'input file permet d'uploader un fichier, le plus souvent de données.

```
fileInput(inputId, label,
          multiple = FALSE,
          accept = c(
            "text/csv",
            "text/comma-separated-values,text/plain",
            ".csv"),
          width = NULL, buttonLabel = "Browse...",
          placeholder = "No file selected")
```



Exemple : file_input.R

Widgets de contrôle

shinyWidgets

Le package `shinyWidgets` permet de choisir d'autres couleurs et d'autres styles de widgets de contrôle.

jQuery knob example:



Exemple : `shinywidgets_input.R`

Ajout d'outputs dans l'UI

Après avoir configuré les éléments d'inputs, il faut ajouter les éléments d'output dans l'UI.

Ces outputs peuvent être de toute nature :

- graphiques
- tableaux
- textes
- etc.

Il faut bien comprendre qu'à ce stade, on ne décide que de **l'emplacement** et de **l'identifiant** des outputs, car ceux-ci n'existeront réellement que lorsque l'étape du server sera terminée.

Comme pour les inputs:

- il existe une fonction shiny pour chaque type d'output,
- chaque outputId doit être unique.

Ajout d'outputs dans l'UI

render() vs output

Les fonctions d'output à mettre dans la partie ui (et les fonctions d'output correspondantes à mettre dans le server) sont:

- `verbatimTextOutput()` / `renderPrint()` pour le texte à la façon de la console R
- `textOutput()` / `renderText()` pour des chaînes de caractères
- `plotOutput()` / `renderPlot()` pour les graphiques
- `tableOutput()` / `renderTable()` pour les tableaux de données
- `imageOutput()` / `renderImage()` pour les images
- `uiOutput()` & `htmlOutput()` / `renderUI()` pour mettre en forme, notamment avec des balises HTML

Le principe est qu'un output ui sera automatiquement mis à jour quand un input de sa fonction render() sera mis à jour sur le server.

Les fonctions d'output pour le texte

Il existe plusieurs façons d'imprimer du texte qui sont illustrés ci-dessous :

```
ui <- fluidPage(verbatimTextOutput("renderprint"),
  verbatimTextOutput("rendertext"),
  textOutput("rendertext1"))

server <- function(input, output, session) {
  output$renderprint <- renderPrint({
    print("Ceci est un output avec renderPrint + verbatimTextOutput"))
  output$rendertext <- renderText({
    "Ceci est un output avec renderText + verbatimTextOutput"})
  output$rendertext1 <- renderText({
    "Ceci est un output avec renderText + textOutput"})}
```

```
[1] "Ceci est un output avec renderPrint + verbatimTextOutput"
Ceci est un output avec renderText + verbatimTextOutput
```

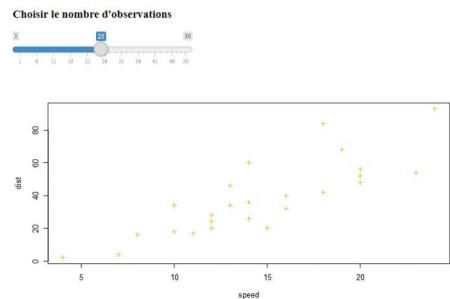
Ceci est un output avec renderText + textOutput

Exemple : textOutput.R

Les fonctions d'output pour les graphiques

La fonction `plotOutput` permet d'afficher un output graphique réactif :

```
ui <- flowLayout(
  sliderInput("slider", label = h3("Choisir le nombre d'observations"), min = 1, max = 50, value = 25),
  plotOutput("plot"))
server <- function(input, output) {
  output$plot <- renderPlot({
    cars2 <- cars %>% sample_n(input$slider)
    plot(cars2, pch=3,col = "Orange"))})
```



Exemple : `plotOutput.R`

Les fonctions d'output pour les tableaux

La fonction `tableOutput` permet d'afficher un output tableau réactif :

```
ui <- fluidPage(fluidRow(column(12,
  tableOutput('table'))))
server <- function(input, output) {
  output$table <- renderTable({
    iris2})}
```

Choisir l'espèce pour afficher 5 observations

setosa
 versicolor
 virginica

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
4.60	3.10	1.50	0.20
4.90	3.10	1.50	0.10
4.80	3.00	1.40	0.30
5.00	3.50	1.60	0.60
5.10	3.80	1.90	0.40

Exemple : `tableOutput.R`

Les fonctions d'output pour les images

La fonction `imageOutput` permet d'afficher un output image réactif :

```
ui <- fluidRow(
  radioButtons(
    column(4,      imageOutput("image2"))
server <- function(input, output) {
  output$image2 <- renderImage({
    if(input$radio == "setosa"){
      list(src = "images/iris_setosa.jpg", height = 240, width = 300)
    }
  })
}
```

setosa
 versicolor
 virginica



Exemple : `imageOutput.R`

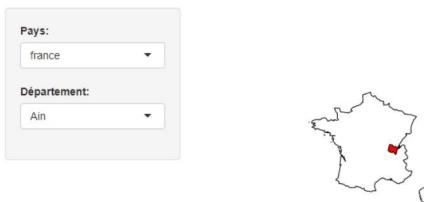
Les fonctions d'output UI / html

htmlOutput est une fonction particulière, car elle permet de générer une ui dynamique : autrement dit, elle permet de créer des objets réactifs qui dépendent de l'input de l'utilisateur.

Ceci permet par exemple de zoomer au sein d'une zone géographique.

```
ui <- fluidPage/sidebarPanel(      htmlOutput("country_selector"))

server <- function(input, output) {
  output$country_selector = renderUI({
    selectInput(inputId = "state", label = "Pays:",
               choices = as.character(unique(df$country)), selected = "france") })}
```

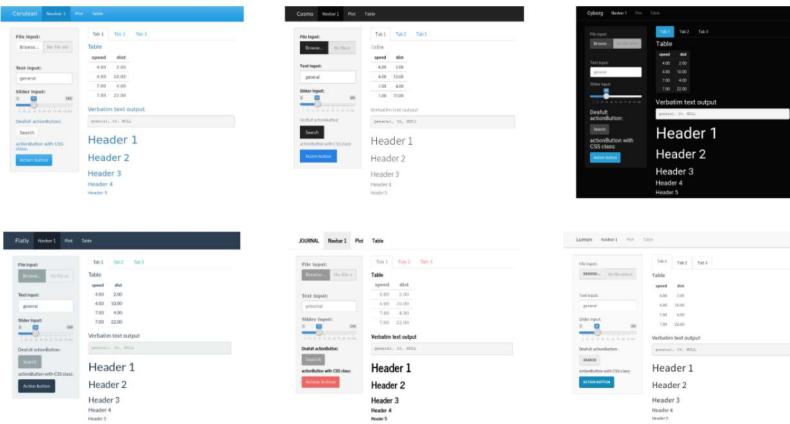


Exemple : htmlOutput.R

Les thèmes Shiny

Le thème par défaut des applications Shiny est dérivé du framework [Bootstrap](https://getbootstrap.com) (<https://getbootstrap.com>).

La librairie `ShinyThemes` (<https://rstudio.github.io/shinythemes>) permet d'utiliser 16 thèmes prédéfinis :



Exemple : `shinythemes_demo.R`

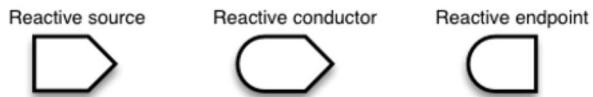


La construction du server

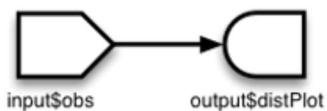
Retour sur la réactivité

(<https://shiny.rstudio.com/articles/reactivity-overview.html>) Dans Shiny nous avons 3 classes d'objets réactifs :

- les sources réactives
- les conducteurs réactifs
- les terminaux réactifs

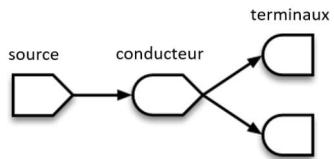


La plus simple des relations entre une **source** et un **terminal** est décrite dans le schéma suivant :



Le conducteur réactif

Si l'on multiplie les sources et les terminaux, on peut avoir besoin d'un objet intermédiaire : le conducteur réactif



Une façon de voir ces 3 classes d'objets Shiny est d'utiliser le concept de filiation:

- un conducteur réactif peut à la fois dépendre d'autres objets et avoir des dépendances en même temps ("parent" et "enfant")
- une source n'a que des dépendances ("parent")
- un terminal ne peut que dépendre d'autres objets ("enfant")

Valeurs, expressions et observateurs

Les **valeurs réactives** contiennent tout simplement des valeurs qui peuvent être lues par d'autres objets réactifs. C'est l'**input** qui est sélectionné par l'utilisateur, ou bien par exemple selon le temps qui s'écoule.

L'implémentation de la notion de **conducteur réactif** est appelée **expression réactive**.

Cela sert de cache mémoire lorsque l'utilisateur change l'**input**, lors d'actions telles que:

- l'accès à une base ou un fichier de données
- l'exécution d'un calcul gourmand en ressources

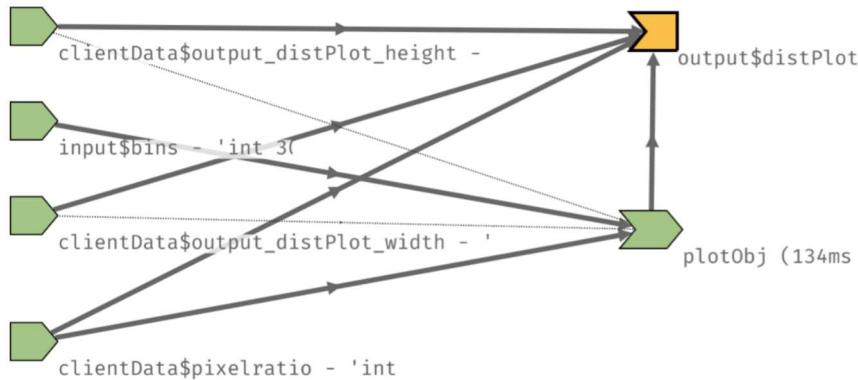
Enfin, les **terminaux réactifs** sont autrement appelés **observateurs**.

Ils accèdent aux valeurs et conducteurs réactifs. Leur rôle est, la plupart du temps, d'envoyer des données vers l'interface (navigateur web).

C'est tout simplement l'**output** du server.

Le package reactlog

Pour illustrer quelques notions que nous venons de voir, le package `reactlog` peut être utile :



Ce package permet en effet de créer un graphique qui représente toutes les interactions entre les éléments réactifs de l'application.

Exemple : `reactlog_demo.R`

La fonction render()

Comme nous l'avons déjà vu, la fonction `render()` est fonction de base pour gérer l'output dans le serveur.

Il en existe plusieurs (`renderDataTable`, `renderPlot`, etc.) selon le type d'output.

Chacune d'entre elles contient comme argument une expression (entre {}) qui peut être plus ou moins complexe.

```
server <- function(input, output) {  
  output$selected_var <- renderText({  
    paste("You have selected", input$var)  
  })}
```

Cette expression est un ensemble d'instructions qui va être lancé en même temps que l'application, puis re-exécuté chaque fois que l'objet est mis à jour.

Autrement dit, l'utilisation de `render()` indique que l'expression qui y est contenue ne doit être exécutée que lorsque ses dépendances (généralement `input$`) changent.

La fonction reactive()

Cette fonction permet de s'assurer que:

- l'expression réactive n'est exécutée que lorsque l'input dont elle dépend est modifié
- les résultats du calcul de l'expression sont partagés par l'ensemble de l'application, ce qui implique que ce calcul n'est effectué qu'une seule fois.

```
server <- function(input, output) {  
  datasetInput <- reactive({  
    switch(input$dataset,  
      "rock" = rock,  
      "pressure" = pressure,  
      "cars" = cars) })  
  output$view <- renderTable({head(datasetInput(), n = input$obs)})  
}
```

A noter dans l'exemple : output\$view dépend de 2 inputs, et sera donc actualisé à chaque fois que l'un ou l'autre sera modifié.

Exemple : reactive_function.R

La fonction isolate()

Cette fonction permet de faire calculer un output qui ne soit pas réactif, bien que dépendant de l'input.

Dans l'exemple ci-dessous, du fait de l'encapsulation de l'expression dans isolate(), l'affichage du graphique de distribution ne sera pas ré-exécuté systématiquement à chaque changement de la valeur réactive input\$obs.

Le graphique ne sera actualisé que si l'on clique sur l'actionButton (étiqueté "goButton" dans l'UI).

```
server <- function(input, output) {  
  output$distPlot <- renderPlot({  
    input$goButton  
    dist <- isolate(rnorm(input$obs))  
    hist(dist)  
  })}  
}
```

Il faut comprendre ici que du fait de la réactivité, bien que le vecteur dist soit calculé à chaque changement d'input, le plot n'est affiché qu'au clic.

Exemple : isolate_function.R

La fonction eventReactive()

Nous venons de voir avec la fonction `isolate()` qu'il est parfois nécessaire d'attendre que l'utilisateur prenne une action spécifique (comme un clic sur un `actionButton`) avant d'effectuer un calcul.
Une valeur ou expression réactive qui sert de déclencheur est appelé un event.

Dans l'exemple ci-dessous, `eventReactive()` permet d'afficher les `x` premières lignes de données, à chaque fois qu'on clique sur le bouton "Montrer"

```
server <- function(input, output) {  
  df <- eventReactive(input$button, { head(cars, input$x)})  
  output$table <- renderTable({df()})  
}
```

Autrement dit, `eventReactive()` génère une nouvelle valeur qui ne se met à jour qu'en réponse à un événement.

Exemple : `eventReactive_function.R`

Les modules Shiny

En théorie une application Shiny peut très bien fonctionner en utilisant des fonctions classiques, mais on peut faire face à plusieurs problèmes lors du développement d'une application plus complexe, du fait notamment que les ids d'input et output doivent être uniques.

Une solution adoptée par beaucoup de langages de programmation (Java, C++, etc.) consiste à créer des **espaces de noms** ("namespace" en anglais) qui auront des noms distincts mais peuvent contenir des ids identiques entre espaces.

Dans le contexte de Shiny, on appelle ces **espaces de noms** des **Shiny modules**, qui ne sont en fait que des morceaux d'application Shiny, dans le sens où on ne peut les exécuter seuls.

Ces modules Shiny peuvent:

- remplacer n'importe quelle partie d'une application (input, output)
- être ré-utilisés entre applications.

Nous ne décrirons pas ici les particularités dans la programmation de ces modules : plus de détails sont disponibles ici (<https://shiny.rstudio.com/articles/modules.html>) ou encore là (https://idc9.github.io/stor390/notes/shiny.html#shiny_modules).

Exemple : `shiny_modules.R`

Annexe

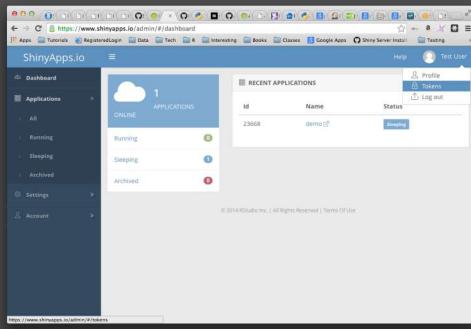
Utilisation d'un fichier global.R

Dans beaucoup d'applications, vous allez avoir besoin de faire quelques opérations sur les données telles que :

- le chargement du fichier source (type .csv)
- la connexion à une base de données
- le chargement de packages
- la définition de fonctions spécifiques
- la définition de constantes
- la mise en forme des données
- etc.

Un moyen de centraliser ces opérations en dehors de l'application elle-même est de créer un fichier `global.R` qui sera hébergé dans le même répertoire que le reste de l'application (`app.R` par exemple).

Cette méthode a l'avantage d'améliorer la rapidité d'exécution, car le script ne s'exécute qu'une seule fois au lancement de l'application et créé des variables et fonctions globales.



Déploiement des applications

Déploiement en local

La façon la plus simple de partager une application Shiny est de zipper le répertoire contenant tous les éléments de l'application et de l'envoyer aux utilisateurs (par email, usb, etc.).

Il suffit ensuite à l'utilisateur de lancer l'application sur son poste avec la commande `shiny::runApp`.

Cette méthode est rapide, mais a 2 inconvénients:

1. l'utilisateur doit avoir R sur son poste, ainsi que tous les packages utilisés dans l'application
2. toute mise à jour de l'application nécessite de refaire toute la procédure

Une autre façon de déployer en local est de créer une fonction, un package ou un exécutable.
Ces méthodes sont évoquées dans les slides suivantes.



Déploiement sous forme de fonction

Il est tout à fait possible de transformer une application Shiny en fonction R plus classique : il suffit pour cela de paramétriser la fonction `shinyApp`.

Dans l'exemple, nous allons construire une fonction `binner` qui:

- prend en paramètre le nombre de barres `x` que doit comporter l'histogramme
- prend en input un fichier csv comprenant la série de nombres à représenter

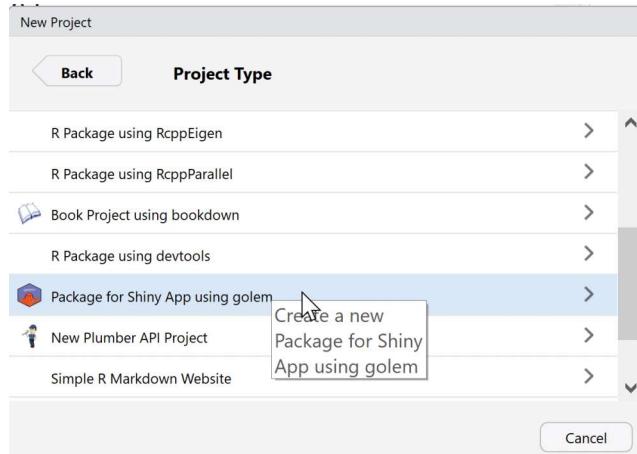
L'output de cette fonction est alors l'histogramme.

Exemple : `shiny_function.R`

Déploiement sous forme de package

Au sein de nombreux packages existent des petits applications Shiny permettant de faire une démo des capacités du package en question. (exemple : package esquisse)

Le package `golem` (voir cette page (<https://thinkr-open.github.io/golem/>)) procure un template pour développer une application sous cette forme.



Déploiement sous forme d'exécutable

le package RInno



L'idée derrière ce package (<https://github.com/ficonsulting/RInno>) est d'encapsuler l'application Shiny au sein d'un exécutable (.exe) grâce au framework [Electron](https://github.com/electron/electron) (<https://github.com/electron/electron>) de GitHub.

```
create_app(  
  app_name      = "MyAppName",  
  app_dir       = "My/app/path",  
  pkgs          = c("jsonlite", "shiny", "magrittr", "xkcd"), # CRAN-Like repo packages  
  include_R     = TRUE,    # Download R and install it with your app, if necessary  
  R_version     = "2.2.1",  # Old versions of R  
  default_dir   = "pf")    # Install app in to Program Files
```

Déploiement sous forme de container



Dans la continuité de la mise en package évoquée plus haut, l'utilisation de **Docker** permet de constituer une image, constituant alors **un environnement isolé du reste de la machine qui contient tout ce qu'il faut pour faire tourner une application**.

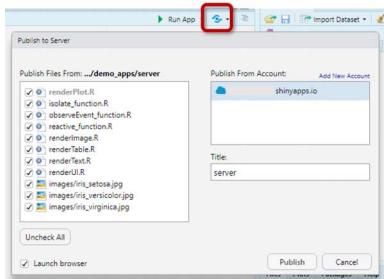
Autrement dit, on peut alors distribuer l'application Shiny sans se préoccuper de la configuration de la machine de l'utilisateur.

Cette page (<https://thinkr.fr/dockeriser-application-shiny/>) décrit le processus, qui est cependant assez complexe pour quelqu'un qui ne serait pas à l'aise avec les notions avancées de programmation.

Déploiement en ligne

Shinyapps.io

Ce site web (<https://www.shinyapps.io/>) permet en quelques clics, sans quitter Rstudio, de publier une application en ligne. Exemple ici (<https://alexkruse.shinyapps.io/stadtrad/>)



Il existe une version gratuite ayant l'inconvénient de rendre publique votre application.

Il faut compter environ 90€ par mois pour en sécuriser l'accès.

Le site permet d'administrer les applications en ligne :

- en leur allouant plus ou moins de ressources
- en surveillant les statistiques d'accès

Déploiement en ligne

Rstudio.cloud

Cet autre site web (<https://www.rstudio.cloud/>) permet d'utiliser RStudio sans l'installer sur son poste, depuis un navigateur web type Chrome.



Le site permet donc d'héberger et faire tourner un script en ligne, comme s'il était en local.

Il suffit de partager un lien (exemple ici (<https://rstudio.cloud/project/596222>)) et de demander à l'utilisateur de lancer le script avec la commande "Run Document" pour faire une démo.

Déploiement sur un Shiny Server



Shiny Server existe en 2 versions :

- la version **Open Source** (gratuite) qui peut être installée sur n'importe quel serveur
- la version **Pro** (payante) qui dispose de beaucoup plus de fonctionnalités (sécurisation, logs, scheduler, etc.)

RStudio Connect permet également de publier des applications Shiny en ligne, parmi d'autres fonctionnalités collaboratives bien plus étendues.



Shinyproxy est une solution complémentaire permettant de mieux faire face aux challenges induits en entreprise par le déploiement d'une application multi-utilisateurs de façon sécurisée (LDAP, TLS, etc.).
C'est une technologie basée sur Java et Docker.

Il existe aussi le package (<https://github.com/AndreaCirilloAC/ramazon>) ramazon qui propose d'installer une application sur le cloud Amazon.
De multiples tutoriels en lignes expliquent comment déployer sur d'autres providers, mais cela reste relativement complexe pour les néophytes.



L'écosystème Shiny

Les documents et présentations Shiny

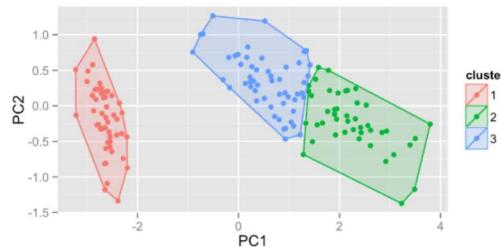
Un phénomène notable dans le domaine de l'analyse de données ces dernières années est l'apparition de publications au format notebook.

Ce format permet au sein d'un même document d'afficher l'analyse et le code :

- plan et commentaires,
- code résultats du code.

```
Specifying frame = TRUE in autoplot for stats::kmeans and cluster::* draws convex for each cluster.
```

```
autoplot(fanny(iris[-5], 3), frame = TRUE)
```



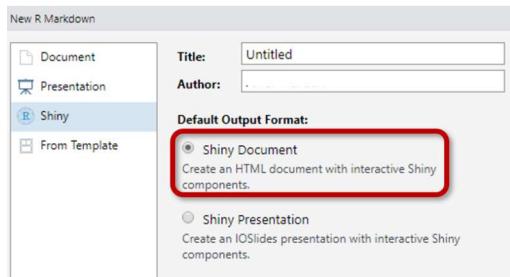
Shiny permet d'ajouter une autre dimension intéressante : l'interactivité.

Documents Shiny

Intégration au RMarkdown

Le R Markdown est le langage de base des notebooks. Il permet grâce à un système de balises (comme pour le HTML) de mettre en forme du texte, et de générer un document dans un format de votre choix (html, word, pdf, etc.).

Dans RStudio, on peut créer un document html avec des composants Shiny en faisant File -> New File -> R Markdown, en finissant par le choix **Shiny Document**



Exemple : shinydoc_demo.Rmd

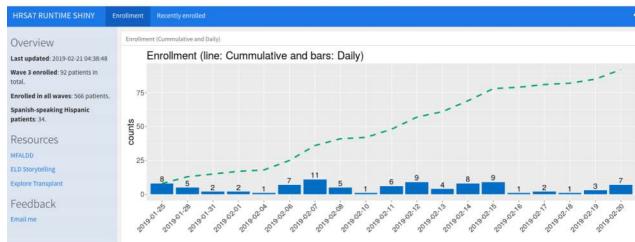
Les dashboards

Shiny permet, couplé à du RMarkdown, de générer des dashboards interactifs de qualité tout à fait comparable à de nombreux autres outils BI type **Qlik**, **Power BI** ou **Tableau**.

La différence essentielle tient au fait que la construction du dashboard se fait en codant avec les langages R et RMarkdown plutôt qu'avec une interface clic-bouton.

Les 2 packages les plus utilisés abordent le codage du dashboard de façon différente:

- **shinydashboard** est une sorte d'extension du package **shiny** avec des éléments d'UI spécifiques (comme par exemple **dashboardSidebar**)
- **flexdashboard** est un template R Markdown adapté à la création de dashboards



Dashboards

Shinydashboard

shinydashboard se compose avec du pur code shiny, mais avec quelques fonctions spécifiques :

```
library(shinydashboard)

dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody(tabItems(
    tabItem(tabName = "Onglet",
      fluidRow()
  )
)
```

Exemple : shinydashboard_demo.R

Dashboards

Flexdashboard

flexdashboard se compose comme un document **R Markdown**, avec une trame déterminée par des mots clés tels que `Column`, `{.sidebar}` ou encore `{.tabset}`

Il suffit d'inclure `runtime: shiny` dans l'entête du script pour pouvoir y inclure des éléments Shiny.

```
Column {.sidebar}
-----
```{r}
selectInput("pays",
 label = "Quel pays?",
 choices = choix.pays,
 selected = "Belgique")
```

Column {.tabset}
-----
### Carte
```{r}
renderLeaflet()
```

```

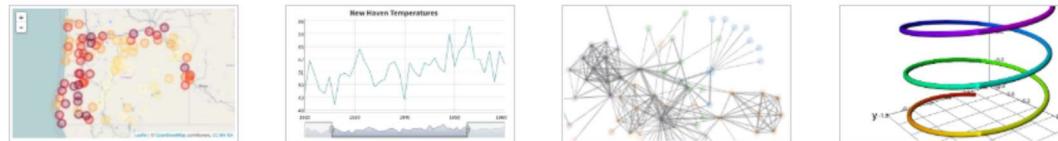
Exemple : `flexdash_demo.Rmd`

Les htmlwidgets

Les HTML widgets (https://www.htmlwidgets.org/showcase_leaflet.html) peuvent être utilisés dans R à partir d'un simple script comme depuis du R Markdown ou au sein d'une application Shiny.

Parmi les 107 widgets (<http://gallery.htmlwidgets.org/>) disponibles à ce jour, nous allons passer en revue:

- des widgets de **dataviz** : Plotly, Highcharter, Bokeh et Dygraphs
- des widgets de **tableaux de données** : DT et rpivotTable
- des widgets de **cartographie** : Leaflet et mapview

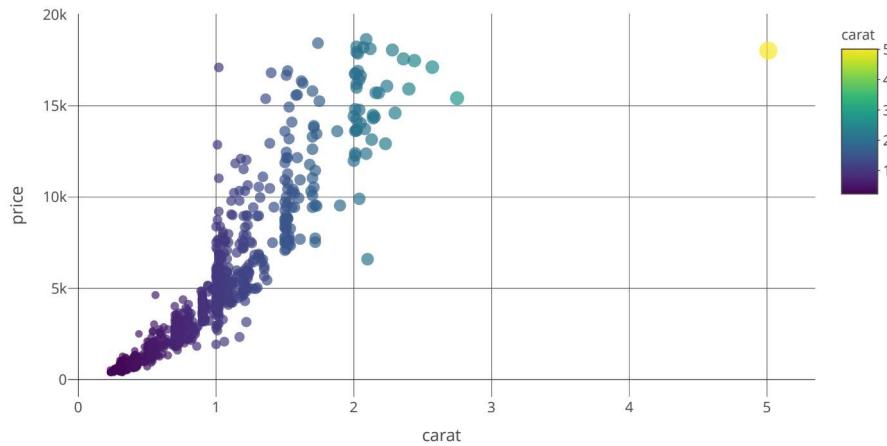


Enfin, nous évoquerons crosstalk qui est un add-on permettant de faire interagir les widgets.

Htmlwidgets de dataviz

Plotly

Plotly peut fonctionner de 2 façons: avec une syntaxe propre, ou en convertissant un objet ggplot2.

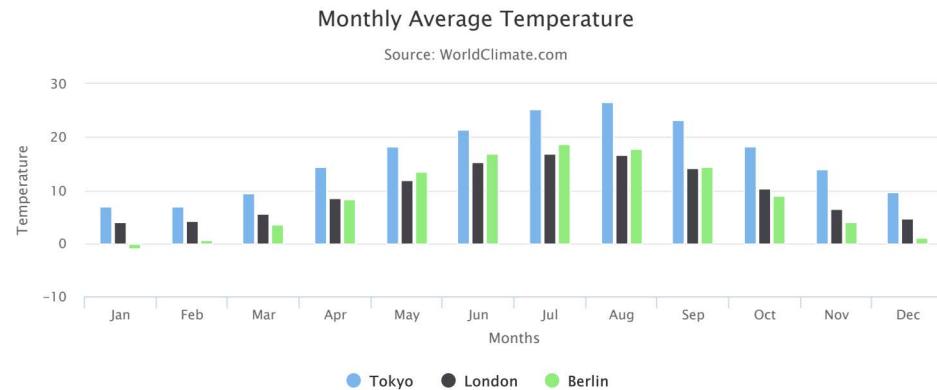


Exemple : `plotly_demo.R`

Htmlwidgets de dataviz

Highcharter

Highcharts est à l'origine une librairie javascript dont la principale qualité est la consistance entre les différents types de graphs, ainsi que leur simplicité.

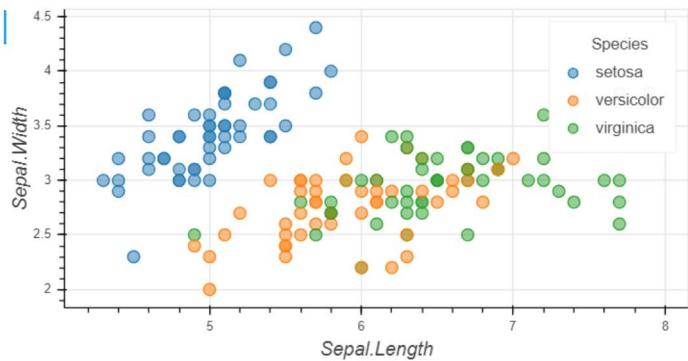


Exemple : `highcharter_demo.R`

Htmlwidgets de dataviz

Bokeh

Bokeh est une librairie de dataviz qui a des interfaces en Python, Scala, Julia et R.



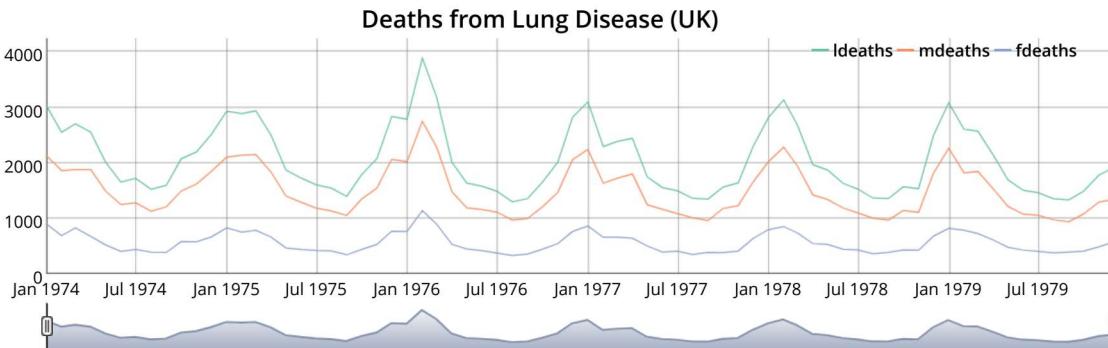
Exemple : bokeh_demo.R

Htmlwidgets de dataviz

dygraphs

dygraphs est à l'origine une librairie JavaScript open source permettant de tracer des séries temporelles.

La librairie dygraphs reconnaît en particulier automatiquement les objets R au format xts. L'exemple ci-dessous ajoute un curseur pour sélectionner un intervalle de temps en abscisse :



Exemple : dygraphs_demo.R

Htmlwidgets de données

DT

DataTables est à l'origine un plug-in pour la librairie JQuery permettant d'afficher des tableaux de données interactifs.

La librairie DT offre une interface R à ce plug-in, avec la possibilité de filtrer, paginer, trier les données ou encore formatter les cellules.

| Search: <input type="text"/> | | | | | | | | | | | |
|------------------------------|------|-----|------|-----|------|-------|-------|----|----|------|------|
| | mpg | cyl | disp | hp | drat | wt | qsec | vs | am | gear | carb |
| Mazda RX4 | 21 | 6 | 160 | 110 | 3.9 | 2.62 | 16.46 | 0 | 1 | 4 | 4 |
| Mazda RX4 Wag | 21 | 6 | 160 | 110 | 3.9 | 2.875 | 17.02 | 0 | 1 | 4 | 4 |
| Datsun 710 | 22.8 | 4 | 108 | 93 | 3.85 | 2.32 | 18.61 | 1 | 1 | 4 | 1 |
| Hornet 4 Drive | 21.4 | 6 | 258 | 110 | 3.08 | 3.215 | 19.44 | 1 | 0 | 3 | 1 |
| Hornet Sportabout | 18.7 | 8 | 360 | 175 | 3.15 | 3.44 | 17.02 | 0 | 0 | 3 | 2 |

Exemple : DT_demo.R

Htmlwidgets de données

rpivotable

rpivotTable est un package permettant de manipuler des tableaux de données à la façon des tableaux croisés dynamiques Excel:



The screenshot shows the rpivotTable interface. At the top, there are dropdown menus for "Row Heatmap" (set to "Sum"), "Column Heatmap" (set to "Freq"), and "Value" (set to "Sex"). Below these are buttons for "Sum" and "Freq". The main area displays a heatmap of hair and eye colors. The columns are grouped by Sex (Female and Male) and further by Eye color (Blue, Brown, Green, Hazel). The rows are grouped by Hair color (Red, Black, Blond, Brown). The heatmap values represent frequencies, with some cells highlighted in red. A "Totals" row at the bottom provides summary values for each category.

| | | Freq | | | | | | | | |
|--------|--|---------|--------|-------|-------|--------|-------|-------|--------|--------|
| | | Sum ↓ ↔ | | | | | | | | |
| | | Sex | | Eye | | | | | | |
| Hair | | Sex | Female | | | Male | | | Totals | |
| | | Eye | Blue | Brown | Green | Hazel | Blue | Brown | Green | Hazel |
| Red | | 7.00 | 16.00 | 7.00 | 7.00 | 10.00 | 10.00 | 7.00 | 7.00 | 71.00 |
| Black | | 9.00 | 16.00 | 2.00 | 5.00 | 11.00 | 32.00 | 3.00 | 10.00 | 108.00 |
| Blond | | 64.00 | 4.00 | 8.00 | 5.00 | 30.00 | 3.00 | 8.00 | 5.00 | 127.00 |
| Brown | | 34.00 | 68.00 | 14.00 | 29.00 | 50.00 | 53.00 | 15.00 | 25.00 | 408.00 |
| Totals | | 114.00 | 142.00 | 31.00 | 46.00 | 101.00 | 98.00 | 33.00 | 47.00 | 592.00 |

Exemple : rpivotable_demo.R

Htmlwidgets de cartographie

Leaflet

Leaflet est une des librairies javascript de cartographie les plus connues et les plus utilisées.

Son implémentation dans R fonctionne en ajoutant des couches successives:



Exemple : leaflet_demo.R

Htmlwidgets de cartographie

Mapview

Mapview fonctionne comme une simplification de leaflet.

Par conséquent, les possibilités y sont moindres mais cela suffit pour construire des cartes rapidement.



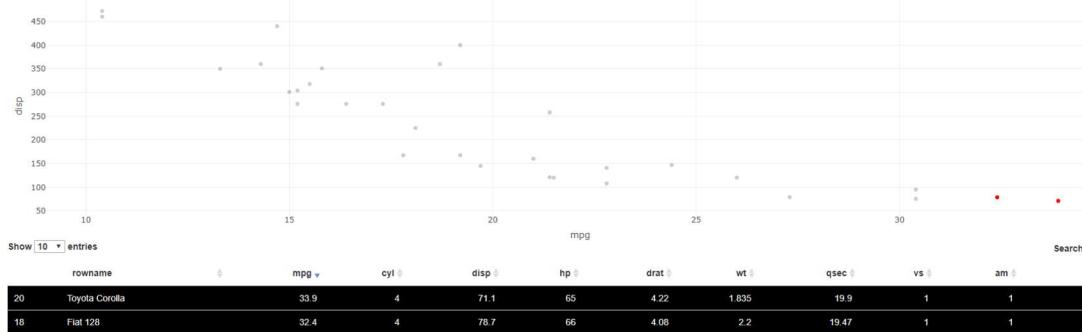
Exemple : `mapview_demo.R`

Interactions entre les htmlwidgets

Crosstalk

Nous avons déjà vu comment mettre en évidence sur un graphique la sélection d'une ou plusieurs observations d'un tableau de données ou d'un autre graphique à l'aide des **modules Shiny**.

crosstalk permet de créer ces interactions entre différents htmlwidgets, ici avec DT et plotly :



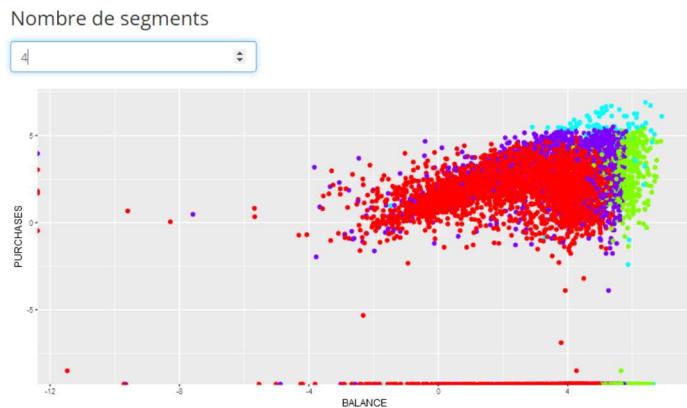
Exemple : crosstalk_demo.R

Exercices

Exemple de restitution Machine Learning

Pour beaucoup de data scientists, la restitution d'un modèle de Machine Learning auprès des métiers peut s'avérer problématique : la création d'une simple application Shiny peut souvent répondre à ce besoin.

Dans cet exemple on peut visualiser les résultats d'une segmentation à l'aide de **kmeans**.



Exemple : ml_demo.R

Exercice final : à vous de jouer!



Instructions: [exercice_final.html](#)