ShinyTex Manual

Version 0_0.11

Howard Seltman

July 8, 2016

Table of Contents

Introduction

ShinyTex is a system for authoring interactive World Wide Web applications (apps) which includes the full capabilities of the R statistical language, particularly in the context of Technology Enhanced Learning (TEL). It uses a modified version of the LaTeX syntax that is standard for document creation among mathematicians and statisticians. It is built on the Shiny platform, an extension of R designed by RStudio to produce web apps.

The goal is to provide an easy to use TEL authoring environment with excellent mathematical and statistical support using only free software. ShinyTex authoring can be performed on Windows, OS X, and Linux. Users may view the app on any system with a standard web browser.

An example of a shinyTex app can be found at JoelChiSquare. Some key features that you will note if you run that app are tabbed pages for different components of the TEL "lesson"; sets of multiple choice questions with corresponding detailed responses for each answer; use of complex mathematical typesetting; plots and tables; use of tooltips for important key words and phrases; hyperlinks to other web apps; and interactive, live R demonstrations with user controlled inputs. The corresponding shinyTex authoring file is found at JoelChiSquareTab.tex, which is a plain text file with a ".tex" extension. The syntax is described in detail below, but a

LaTeX user will note that it consists of standard LaTeX code with added Shiny-related environments and a few new commands.

The shinyTex system will be of use to anyone wishing to author statistical or mathematical TEL. It is designed for quick learning by those familiar with LaTeX, but should also be of use to those who do not know LaTeX because it uses a fairly small subset of LaTeX, and a tutorial is provided here. A working knowledge of R is assumed, but not of Shiny. The required subset of Shiny knowledge is all explained here. Equivalent functionality may be achieved using only Shiny, but the authoring process is much more difficult, and requires additional understanding of the technical details of Shiny and HTML. On the other hand, shinyTex necessarily provides less than full Shiny capabilities, but you should feel free to contact the author if you would like to suggest any additional capabilities.

The website for shinyTex is http://www.stat.cmu.edu/~hseltman/shinyTex/.

Quick Start

Conventions of this manual

This manual uses the camel-case variable naming system. It also uses the convention that names that start with "mu", e.g., "myBank", are a clue that when you are writing your authoring code you must make up your own name to replace the "myBank" name. Names like "someEnvironment" indicate that one of several specific choices is possible. Any R, LaTeX, or shinyTex code is written in `a font like this`.

Overview of the Authoring Process

The authoring process consists of writing specialized LaTeX code in a LaTeX file (a plain-text file that follows the conventions of LaTeX and has the file extension ".tex") along with the necessary R code inside of special begin/end markers (called "environments" by LaTeX). The authoring text may be created with any standard text editor capable of using a plain ASCII format. An obvious choice is whatever editor you use for R or RStudio. (In the future, a shinyTex LaTeX document class will be provided for those who wish to start the authoring process in LaTeX and for the task of creating a single printable version of the app for editing purposes.)

The actual process of creating the app from the LaTeX file must be performed in R or RStudio. To create the app, e.g., for a lesson in a file called "myApp.tex", you simply run the R command `shinyTex("myApp")`. The result will either be one or more detailed error messages explaining what mistakes you made or the production of several files (ui.R, server.R, some #-#.html files, and a myBankQuiz.RData file for each quiz bank you create) which together constitute a complete, stand-alone Shiny app.

To run the Shiny app created by shinyTex during the authoring process, you will use the R/Shiny command `runApp()`. (Note that your users will only need to go to a particular URL in their browser.) While examining your app in your browser, you will usually note aspects of the app that require correction or improvement, and then you will close the app and repeat the authoring cycle, i.e., edit your LaTeX file, re-run shinyTex (), and re-run runApp ().

To deploy the app, you will probably need to consult with your local computer networking professional. Deployment is the same as for any Shiny app, and has nothing to do with shinyTex. The best choice is a Shiny server at your institution. Alternatives are use of the RStudio server via shinyapps.io (which has a limited use free version, and pay versions which allow higher levels of use), or distributing the app files to your users, e.g., through GitHub (https://github.com) or by simply sending the files to your users. The latter approach requires your users to load R and Shiny on their computers.

Setup

This section describes how you must setup your computer to use shinyTex. If you deploy your app only through sharing files with users rather than using a Shiny server, they will need to perform much of this same setup (see "Minimal setup for users when a Shiny server is not used", below).

For all authoring you must install a recent version of R (if you have not updated R in the last six months or so, you should consider doing so now). Many users find RStudio a more convenient and efficient way to use R, and it is suggested, but not required, for shinyTex. (Again, if you have not updated recently, please do so now.) ShinyTex requires the R packages "shiny" and "stringr". Install and/or update them now.

The functionality of shinyTex is split between the R package "shinyTex" (current versions) which is used only in the authoring process, plus an R file and a css file which are needed by Shiny when the app is run by you or your users. Install shinyTex from the zip file (from the RStudio "Packages" tab, click "Install", then change "Install from:" to "Package Archive File", click "Browse", find your copy of the zip file, and click "Install". The installation only needs to be performed once. Then each time you start R, you need to run the `library(shinyTex)` command (or use a .First file – see here). The first time you start shinyTex in each directory you use, you need to run `setupShinyTex()` to setup for running apps. This step copies "shinyTexAux.R" to your current directory and "shinyTex.css" to a "www" subdirectory.

For deployment, the files required to be in the deployment directory are shinyTexAux.R, shinyTex.css in a "www" subdirectory, and the following files produced by shinyTex: ui.R, server.R, any #-#.html files, and any myBankQuiz.RData files (one for each quiz bank).

A simple app

Here are the steps for creating a simple first app, excluding the step of working in LaTeX, and assuming that you are using RStudio (a similar process is followed for stand-alone R).

A. **Create your app folder**: Create a folder. Start RStudio and (recommended, but not required) create a new project in the chosen folder using "File / New Project… / Create project from Existing Directory" on the RStudio menus. If you have not already done so, under the Packages tab in RStudio, choose Install, and install "shiny" and "stringr" from CRAN.

B. **Install shinyTex**: Run `library(shinyTex)`, and then run `setupShinyTex()`.

C. **Load a sample app authoring file or start one from scratch**: You can use "File / New File / Text File" on the R menus to create an authoring file from scratch. More likely, you may want to load the shinyTexTemplate.tex file or another existing authoring file and edit

it. For now, download simpleApp.tex. Use "File / Open File" from the RStudio menus to open the file. Spend a few minutes looking at the contents, including the comments (denoted with "%", as is required in LaTeX).

D. **Create the app from the authoring file**: Enter the R command `shinyTex()` to convert the LaTeX file into a shiny app. This is sometimes called "compiling" your app.

   You will notice that when you compile your app, shinyTex gives you a message about successful app creation or a detailed cause of failure. For the simpleApp.tex example, successful app compilation creates server.R, ui.R, 3-1.html, 6-1.html, 6-2.html, 6-3.html and 6-4.html. All these files together (along with shinyTexAux.R, any feedbackFor.R files, any hiddenFeedbackFor.R files, and www/shinyTex.css) constitute the "compiled app" that Shiny reads when you run your app.

   If you have more than one app in the same directory, the most recently edited app will be created unless you supply the app filename as an argument to `shinyTex()`. E.g., you might use `shinyTex("app2")` or `shinyTex("app2.tex")`.

   If you are using RFeedback "checks" (see below), then you have the option to store some or all of those checks in a separate subdirectory. In that case, the checks in the subfolder are run only when you add the argument, e.g., `shinyTex()` runs only the checks in the current directory, while `shinyTex(checkDir="check")` runs the checks in both the current directory and the "check" subdirectory.

E. **Run the app**: Enter the R command `runApp()` to run your app. By default, the runApp command opens the app in the "RStudio viewer window" for quick viewing with some missing features; you will want to next click "Open in browser" to see all of the features. Now you can work with the app. (See "Development Cycle and Debugging" for how to change the default.) Explore both tabs. When you are done exploring close the "A Really Simple ShinyTex App" tab in the browser. Then either click the "Close" in the "RStudio viewer window" or press "Escape" in the RStudio console or click the red "Stop" icon.

F. **Edit the app**: In the RStudio text editor, try making any simple changes to the sample app.

G. **Check out your changes**: Re-run the R command `shinyTex()` to create your new app (all old app files are erased first automatically). If you see any error messages, try to correct your errors and re-run `shinyTex()`. Next re-run `runApp()` to see the effects of your changes.

H. **Deploy your app**: If you have a Shiny server at your institution or if you wish to use shinapps.io, you can deploy your app by uploading the files created plus shinyTexAux.R and www/shinyTex.css. Or you can send these files to anyone who has Shiny set up on his or her computer.

Details of creating the LaTeX authoring file

About the LaTeX authoring file

All LaTeX/shinyTex and R code goes in a single file except for the quiz banks (see below). This file must be a plain ASCII text file with no formatting, which excludes the .docx type produced by Microsoft Word as well as any Rich Text Format (rtf) files. The code is case sensitive.

The suggested extension is ".tex" for compatibility with LaTeX, but any other extension is allowed. If ".tex" is used, it may be left off in the `shinyTex()` call, otherwise the extension is required.

Comment lines may be added anywhere in the authoring file, and are recommended. A comment is any line with a percent sign (%) as the first non-blank character, or all characters after a percent sign, as taken from LaTeX. This means that to include a percent sign in your text, you must use "\%".

Most of the extensions to LaTeX that are unique to Shiny are LaTeX commands and environments that start with "\shiny".

If you are unfamiliar with the conventions of LaTeX, you should read "LaTeX primer" below. Key points are use of the backslash character (\) to indicate LaTeX command; use of curly braces ("{" and "}") to delimit command options or text to be affected by a command; blocks of text with special meaning/function called "environments" and delimited by lines like "`\begin{foo}`" and "`\end{foo}`"; and use of single or double dollar signs ($) to delimit mathematical formulas coded in the LaTeX math syntax.

Overall structure

Apart from comments, the overall structure of a shinyTex authoring (LaTeX) file is as follows:

- \def lines to specify the author, title, and formatting of the app title
- `\begin{document}`
- all of the text that constitutes the authoring
- `\end{document}`

The "text that constitutes the authoring" has one of three overall formats depending desired on the overall look of your app. This look is defined use one of these "outer" environment setups.

For a **simple, unstructured single-page app**, the main app code goes inside of the lines "`\begin{shinySinglePage}`" and "`\end{shinySinglePage}`".

For a **simple single-page app divided vertically into a side panel and a main panel**, the authoring text is of this format:

```
\begin{shinySinglePage}
\begin{shinySidePanel}
  ... code for the side panel goes here ...
\end{shinySidePanel}

\begin{shinyMainPanel}
  ... code for the main panel goes here ...
\end{shinyMainPanel}
\end{shinySinglePage}
```

For a **tabbed app**, which is most flexible type, the authoring text is of this format:

```
\begin{shinyTab}{tab #1's tab label}
  ... code for first tab goes here ...
\end{shinyTab}

...

\begin{shinyTab}{last tab's tab label}
  ... code for last tab goes here ...
\end{shinyTab}
```

In addition to the use of one of the three formats above, your app can include "quiz banks" that contain sets of questions each of which contain question text, answers, and explanatory responses for each answer. These go in separate ".tex" files (see below).

Inner environments: All shinyTex authoring code is written inside of shinySinglePage, shinyTab, and/or shinyQuizBank "outer" environments. (The shinySidePanel and shinyMainPanel environments, while actually occurring within a shinySinglePage environment, are also considered to be "outer" environments, too.) Within these "outer" environments further code is either written inside of further "inner" environments (e.g., verbatim, tabular, shinyPlot, shinyMCQ, etc.), or written directly ("outside of inner environments").

Text outside of inner environments

Much of your app may consist of explanatory text. In this kind of text one space between words is no different from multiple spaces, and you can include line breaks wherever you find it most convenient. Because the app is deployed as HTML, regardless of spacing all text up to the next blank line is assembled into a

paragraph, and in the app the line breaks depend only on the user's font size and window width. Note that there is no difference between one blank line between paragraphs and multiple blank lines. If you want extra spacing, see the "skip" commands below.

In addition to ordinary text in paragraphs, you may include HTML headers, tooltips (highlighted text that shows additional explanatory text when hovered over with the mouse), web links, and combination "tooltip/web links". In addition, formulas written using the math mode of LaTeX (either within a text line or separated in "LaTeX display mode") are included as "text outside of inner environments". Markup of text such as bolding, italics, and "small-caps" is another form of text outside of inner environments. Finally commands for extra vertical spacing and production of a horizontal rule are considered here.

To place an HTML header in your document use a command like `\h3{My header text}`. The size of the header is controlled by the number, which can be any value from 1 to 4, with 1 being the largest.

If you want extra space between paragraphs or between other shinyTex elements, you may enter the commands `\smallskip`, `\medskip`, or `\bigskip`. These should be entered on their own lines. In addition, you can make a more prominent separation between vertical sections of your app by using `\hr` (again, on its own line), which generates a "horizontal rule", as defined by HTML.

To make some horizontal space between text elements (including links), use the `\spaces{#}` command, where you fill in the desired number of spaces in the place of "#".

To mark text as boldface ("strong" in HTML) use the syntax `\textbf{my text to be bolded}`. (The deprecated LaTeX syntax `{\bf my text to be bolded}` is also allowed.) Similarly `\textem{}` is used for emphasis (italics), and `\textsc{}` is use for small capital letters. ShinyTex uses either `\textbfem{}` or `\textembf{}` for bold italics.

The syntax `\code{my text}` is intended to be used to set off computer code (when part of a sentence rather than in a `\verbatim` environment) by using some combination of a different font, color and/or background color. You can change the default values of these three for your entire app by setting the corresponding "defs" statements in the Latex preamble of the document (see "Starting a new app", below).

Any LaTeX math mode formulas may be entered using $myFormula$ for inline formulas, and $$myFormula$$ for offset formulas (called "display mode" in LaTeX). To include a dollar sign in your text, use "`\$`" to avoid confusion with formulas. Actually, LaTeX formulas in Shiny are handled by MathJax, so you may need to check there for some minor limitations.

To create a "tooltip" for your app, use the syntax `\tooltip{my visible text}{my additional text}`. The "my visible text" will show in your app and it

will be boldfaced. When your user hovers her mouse over the text, the "my additional text" will become visible. This is useful, for example, for definitions.

To create a web link (hyperlink reference), use `\href{my visible text}{myURL}`. The text will be underlined, and when the user clicks, they will be transferred out of the app and to the specified Uniform Resource Locator. An equivalent syntax is `\shinyLink{my visible text}{myURL}`. Alternatively, you can use `\shinyLinkNew{my visible text}{myURL}`, which will open the link in a new window rather than replacing the app in the current window.

To create a web link with a tool tip, use the syntax `\tooltipHref{my visible text}{URL}{my additional text}`. This will create a web link that shows the additional text when the user hovers over the link.

To create a "mail to" linkip, use the syntax `\shinyMailto{my visible text}{email address}{subject}`. This will create a web link that opens the user's mail program and starts a new email to "email address" with the subject text being "subject".

In addition to the above, you can place "input controls" on your app by using the one-line shinyTex commands `\shinySlider{options}`, `\shinyActionButton{options}`, `\shinyCheckbox{options}`, and `\shinyRadioButtons{options}`. See below for details.

The verbatim environment

If you want to enter any text in the exact form you write it, you may place the text inside of a verbatim environment. (This implements the usual verbatim environment in LaTeX, and becomes a <pre></pre> HTML section on the app's web page.) An example is

```
\begin{verbatim}
As verbatim, this text has each line break,
Just where I want it for goodness sake.
\end{verbatim}
```

Tables: The tabular environment

The tabular environment of shinyTex is used to create tables. (It implements a subset of the LaTeX "tabular" environment as an HTML "table".) Currently the syntax is anything similar to this:

```
\begin{tabular}{}
              & column header #1 & column header #2 \\
row header #1 & data             & data \\
row header #2 & data             & data \\
\end{tabular}
```

Further features will be added later.

Lists: The enumerate and itemize environments

Numbered and bulleted lists are implemented as in standard LaTeX. (They become HTML <ol> and <ul> elements in your app.) Nested lists are allowed.

The syntax is anything similar to this:

```
\begin{enumerate}
  \item first item
  \item second item
\end{enumerate}
```

To make bulleted lists instead of numbered lists, substitute "itemize" for "enumerate".

Working with R code in a shinyTex app

A key feature of Shiny and shinyTex is the incorporation of "live" R code, usually with user controllable input and text and/or graphics output. One example is simply demonstrating some R analysis steps. Another common use of live code in TEL is data simulation under various conditions and corresponding EDA and/or model fitting and/or model diagnostics. One may also want to generate many simulated data sets for a given set of conditions, fit one or more models to each, and then present the resulting sampling distributions in some suitable format. Another example is demonstration of multiple steps of data analysis for a single fixed dataset.

You may find it worthwhile to read the Shiny documentation to get a deeper understanding, but you may also find that the documentation provided here is sufficient and simpler. Knowledge of R at the level of whatever analyses you wish to perform is assumed here.

Shiny provides several "input controls" that the user can interact with to set various quantities, and your R code can "read" the current values. To place an input control in your app you use a shinyTex input control command (see below). Each input control has an "inputId" argument that you set to a variable name of your choosing, and your R code accesses the current value of the control by using the form `input$myInputIdName`. E.g., if you call your control "beta", then `input$beta` in your R code refers to the current value of the beta control. When beta is changed by the user, any of your code that depends on beta will be automatically re-run. Distinct from the other input controls, a `\shinyActionButton` control does not have a useful value, but by referring to that button you can control when code is re-run, e.g., to generate a new simulated data set (see below for more details).

In native Shiny, you must maintain separate files or functions for the user interface and the R "server" code. ShinyTex simplifies this by including the entire app in a single LaTeX file with the R code interspersed with the visible "user interface". All R code is placed inside of special code environments as explained here.

Note that the R code for a shiny app is segmented into functionally independent sections, each of which has a specific purpose, and the variables created in each section are unavailable to the other sections. (This is essentially the same as the having separate R functions.) The code is run automatically as needed based on changing user input.

All R code goes inside one of several environments, including "shinyCode", "shinyCodeOutput", "shinyPlot", and "shinyReactive". The "shinyCode" environment is used to hold R code that is run only once, and any variable definitions in this environment ***are*** available to R code in the other code environments, i.e., they are "global". Two examples of code that might be placed in a "shinyCode" environment is code for defining constants, and code for reading or creating data sets that do not change within the lesson.

More specifically, data may be made available to the app by placing a data file in the working directory and including a standard R command such as `read.csv()` in the app to load the data into an R variable. Your app may also load data directly from a web source using a URL as the argument to `read.csv()` or similar R commands. Simulated data may be generated using any required R commands such as `rnorm()` and `data.frame()`.

The "shinyCodeOutput" environment is used to hold R code that should be run and the output displayed in the lesson. The "shinyPlot" environment holds R code to produce a plot. The "shinyReactive" environment is described below.

Shiny provides mechanisms for isolating code that may be time consuming to run and/or that should only be run at specific times such as code for generating simulated data and/or non-trivial data analysis. As an example, consider an app that simulates 1000 data sets based on several user controlled parameters, runs regressions on each data set, and then displays summary statistics and a histogram of the sampling distribution of a particular regression coefficient chosen by use of a radio button. There might also be a slider controlling the bin size of the histogram. In this case it would be slow and highly confusing to regenerate the simulated data every time the user chooses a different coefficient to view or changes the bin size. One sensible choice is to re-simulate the data only when the user clicks a "re-simulate button". Another possible choice is to re-simulate when any simulation parameters change and/or the user explicitly asks for a re-simulation, but not when the bin size changes or a different parameter is selected.

The kind of functionality described in the preceding paragraph is implemented by what is called a "reactive" function in Shiny. The reactive function(s) compute something and return an R object. When any other code segment needs to access that object it calls the function. But an automatic Shiny mechanism assures that the calling functions gets the resulting object directly without re-running the reactive function, unless one of the inputs directly affecting the reactive function has changed. ShinyTex simplifies the process by using a `shinyReactive` environment (see below) to define each reactive function. As an example, consider two independent parts of your app, each of which need to call a data generation function to obtain some simulated data. The reactive function mechanism assures that both parts see the same simulated data, rather than the second part causing a re-simulation when it calls the reactive function the second time. See the examples for more details.

In some sense, the opposite of a reactive function is the Shiny `isolate()` function. In any code segment of Shiny, use of code like `input$myControlId` causes that code to be automatically re-run if the "myControlId" value is changed by the user. To prevent this from happening in any given R code segment, wrap each use of `input$myControlId` in `isolate()` as `isolate(input$myControlId)`.

ShinyTex has a `shinyCode` environment for R code that is neither a reactive function nor specifically related to a visible app output. In contrast to R variables created inside the other code environments, variable created in this environment are available throughout the app. This may include data loaded from files, setting constants that will be available to other code segments, and defining "helper" functions that will be available to other code segments.

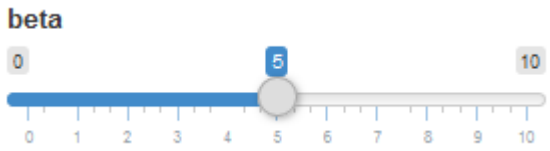
ShinyTex has specific environments for creating several different kinds of R outputs inside your app. As opposed to standard Shiny, shinyTex works by keeping the user interface and R (server) code together in your authoring file, and it hides some of the coding complexities. Generally, simply include standard R code to generate the output. See `shinyPlot`, `shinyCodeOutput`, and others in the documentation below.

Input Controls

ShinyTex input control commands place an input control in your app. The general form is to include `\shinyFoo{inputId=someName; label=someLabel; otherOption1=option1, ...}` as a line in your authoring code at the position where you want the input control to appear. Each control requires "inputId=" and "label=". The inputId is any R variable name that you choose, but each inputId in your entire app must be unique. To refer to an input control in your app, you will use "`input$`" in front of the input id. The text shown in your app adjacent to the input control is whatever you enter between the "label=" text and the next semicolon (";") or the final "}" if no additional options are present. The label text should not include quotes and may include spaces and math formulas within single dollar signs. If a single option requires multiple values, they are separated by a comma (","). See below and/or in the Shiny documentation for the specific options applicable for each input control. Note that any options that have defaults need not be supplied.

Here are the specifics for each input control:

i. `\shinySlider{inputId=myId; label=my label; min=#; max=#; value=#; step=#; animate=FALSE}` creates a slider control that allows a user to set a named quantity to a range of values. It implements Shiny's `sliderInput()` control and the corresponding documentation may be consulted if needed. It looks like this:



Only the most commonly used options are listed here. Of these, only "step=" and “animate=” are optional. The "#" symbol indicates that you should enter a number. "TF" means that you should enter either "TRUE" or "FALSE" (without the quotes and in capital letters). The user will be able to choose any number between "min" and "max" with steps between values equal to "step". The initial value is "value".

ii. \shinyActionButton{`inputId=myId; label=my label`} creates an "action button" control that allows the user to initiate an action such as to generate a new simulation. It implements Shiny’s actionButton() control and the corresponding documentation may be consulted if needed. It looks like this:

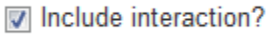
Additionally you can use `textColor=`, `backgroundColor=`, and `borderColor=` to set the colors of the action button to one of the standard CSS colors. If you know about CSS styles, you can use `style=`, but because the semicolon is used to separate options in shinyTex, you must substitute the “|” character for the semicolon when specifying several CSS style elements.

See “Reactive Code” below for more details about the `shinyActionButton`.

iii. \shinyRadioButtons{`inputId=myId; label=my label; choices=my choices; selected=initial value; inline=FALSE`} creates a "radio button" control. It implements Shiny's `radioButtons()` control and the corresponding documentation may be consulted if needed. It looks like this:

The label is placed above the radio buttons. The 'selected=" and "inline=" values are optional. The default is `inline=FALSE` which places the buttons on separate lines rather than next to each other. The button text choices are entered without quotes and separated by commas. The first button is initially selected unless you choose another with “selected=”.

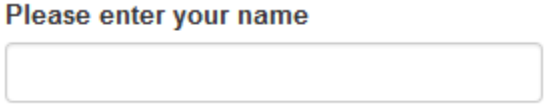
iv. `\shinyCheckbox{input=myId; label=my label; value=TF}` creates a check box control. It implements Shiny’s `checkboxInput()` control and the corresponding documentation may be consulted if needed. It looks like this:



All three arguments are required. The “value” specifies the initial condition

of the control where `TRUE` means checked and `FALSE` means not checked. In your R code, code like `input$myId` represents the current status of the checkbox as a logical R variable.

v. `\shinyTextInput{inputId=myId; label=my label; value=my initial value}` creates a box to enter free text. It implements Shiny's `textInput()` control and the corresponding documentation may be consulted if needed. It looks like this:



You must enter the "inputId" and "label". Do not use quotes. If "value=" is not used, the box starts empty, otherwise it takes your "my initial value" text. In your R code, code like input$myId will represent a character variable holding the user's input.

vi. `\shinyTextInputEnter{inputId=myId; label=my label; value=my initial value}` creates a box to enter free text. It is different from `textInput()` control in that any code that refers to `myId` will not be executed until either the user presses the Enter key or the box loses focus (by Tab or mouse click). In contrast, code containing a reference to a shinyTextInput will be executed repeated, as often as once per typed character. The alternative for plain `shinyTextInput` is that references to it in R code are wrapped in `isolate()`, in which case you will need to use an explicit mechanism such as an "Evaluate" button to trigger evaluation.

You must enter the "inputId". Do not use quotes. If "label=" is used, the label appear to the left of the input box. If "value=" is not used, the box starts empty, otherwise it takes your "my initial value" text. In your R code, code like input$myId will represent a character variable holding the user's input.

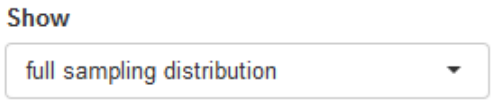
vii. `\shinyTextAreaInput{inputId=myId; value=my initial value; nrows=myRows; ncol=myCols}` creates a multi-line box to enter free text. It implements the `textareaInput()` control that ShinyTex adds to Shiny. It looks like a box with a gray border that is either empty or has the text you specify as "my initial value".

You must enter the "inputId". You may enter an initial value. Do not use quotes for either. The default values are 2 rows and 50 columns. In your R code, code like input$myId will represent a character variable holding the user's input, with one element per line entered by the user.

viii. `\shinyNumericInput{inputId=myId; label=my label; value=my initial value; min=#; max=#, step=#}` creates a box to enter a number. The user may either type the number or use the up/down arrows to change the value. This command implements Shiny's `numericInput()` control and the corresponding documentation may be consulted if needed. It looks like this:

You must enter the "inputId", "label", and "value". Do not use quotes. In your R code, code like input$myId will represent a numeric variable holding the user's input.

ix. `\shinyDropdown{inputId=myId; label=my label; choices=my choices; multiple=TF, selected=value; width=value}` creates a dropdown box to choose among a predetermined set of choices. This command implements Shiny's `selectInput()` control and the corresponding documentation may be consulted if needed. It looks like this:



You must enter the "inputId", "label", and "choices". Do not use quotes. Use commas between the items in your list of choices. "Multiple=" defaults to FALSE, but you can change the value to TRUE to allow multiple values to be selected. The initial value is the first one on the list unless you set another value with "selected=". "Width" can be something like `400px`. In your R code, code like input$myId will represent a numeric variable holding the user's input.

x. `\shinyInsertCode{codeName}` inserts stored code into an `RFeedback` environment. See the `shinyStoreCode` environment below.

Plots: The shinyPlot environment

To generate an R plot and insert it into your app, you place R code inside of a shinyPlot environment. The syntax is

```
\begin{shinyPlot}
... R code to generate a plot goes here  ...
\end{shinyPlot}
```

If you have a reactive object (or several) that you need access to in your plot code, include something like `data = myReactiveFunction()`, where `data` is the name you choose to refer to the data in this environment and `myReactiveFunction` is the name you gave to the reactive code in the `reactiveCode` environment.

Here is a simple example, based on a reactive function named "mySim":

```
\begin{shinyPlot}
  simDat = mySim()  # get simulated data
  plot(simDat$x, simDat$y, xlab="time", ylab="amount",
       type="l", main="Simulated data")
  abline(v=0)
```

```
  text(0, par("usr")[4]-par("cxy")[2], "Start", adj=0)
\end{shinyPlot}
```

The plot will appear in your app at the location of the shinyPlot environment. It will update automatically if `mySim()`'s dependents are changed.

Reactive Code: The shinyReactive environment

As discussed above in "Working with R code in a shinyTex app", reactive code is R code that looks like a function, but is run only when needed, and otherwise returns the results of the previous run. This is typically used to simulate data and to analyze data. The syntax is:

```
\begin{shinyReactive}{myFunctionName}
  … R code …
\end{shinyReactive}
```

The R code that you write does not include the word "function" – it is just lines of R code that produce the reactive object. It is OK (good practice) for the last line of the R code inside the shinyReactive environment to be of the form `return(someObject)`. In the place of "myFunctionName" you enter any valid R variable name that you want, e.g., `theData` or `theModelResults`.

Typically pressing some `\shinyActionButton{}` as well as changing any of the input controls upon which the code depends will "invalidate" the reactive code. Invalidation means that the reactive code will actually be run (again) as well as any functions that call it. Every `input$` object that is mentioned inside the reactive code will cause invalidation of the reactive function. This typically means that, if you have an action button called, e.g., "redo", then the first line of your reactive code is just `input$redo` to assure that the function reacts to pressing than button.

Here is a short example:

```
\begin{shinyReactive}{theModelResults}
  input$redo # react to the redo button
  myData = theData() # run another reactive function
  if (input$interaction) {
    result = lm(y~x1*x2, myData)
  } else {
    result = lm(y~x1+x2, myData)
  }
  return(result)
\end{shinyReactive}
```

Based on this example, you could have a `shinyOutput` environment that starts with `regResult=theModelResults` and then prints `regResult`, as well as a `shinyPlot` environment that includes `regResult=theModelResults` which then uses `resid(regResult)` and `fitted(regResult)` to make a residual vs. fitted plot. With this setup, whenever the "interaction" control (presumably a check box input) changes or any dependents of `theData()` change, any environments that call `theModelResults()` will be automatically updated. Note that if you had used an ordinary, non-reactive function for `theModelResults()`, presumably inside a

`shinyCode` environment, then both the output step and the plotting step would each call that function and it would unnecessarily be run twice. Even worse, if `theData()` were non-reactive, it would have been run twice and the model output and the residual plot would correspond to different simulated datasets!

So the main message is that when simulated datasets are being used, the simulation should be in a reactive function, and if any analyses are at all time consuming, they should also be placed in reactive functions.

Miscellaneous R Code: The shinyCode environment

The `shinyCode` environment, delimited by `\begin{shinyCode}` and `\end{shinyCode}`, holds any R code that does not relate specifically to producing output or for creating reactive functions. It is normally used to read in data from files, define constants, and/or define helper functions. Any arbitrary R code may be included here. (The code will be placed in the server.R file of the shiny app.) As opposed to placed code in the other shinyTex environments, any variables defined in a `shinyCode` environment are available in all other code environments.

Output of arbitrary R code: The shinyCodeOutput environment

Code inside of `\begin{shinyCodeOutput}` and `\end{shinyCodeOutput}` is run and any the output shows both the code and the R results in the app.

For example:

```
\begin{shinyRSession}
  x = c(3, 4)
  sum(x)
\end{shinyRSession}
```

would appear in the app as:

```
> x = c(3, 4)
[1] 7
```

Sometimes you might want to also run some code without showing the user. To do that enclose the entire command in parentheses.

Output of arbitrary R code that looks like an R sesson: The shinyRSession environment

Code inside of `\begin{shinyRSession}` and `\end{shinyRSession}` is run and any output that would normally go to the R console will end up at the current location in the app.

Continuing the example in "Reactive code" above, to see the model results, you could place this code in your app at the position where you want the results to appear:

```
\begin{shinyCodeOutput}
  print(summary(theModelResults()))
\end{shinyCodeOutput}
```

Conditional Panel: The shinyConditionalPanel environment

The `shinyConditionalText` environment can be used to make paragraphs visible only when a particular input control has a particular value. Otherwise the text is hidden. The syntax is:

```
\begin{shinyConditionalText}{inputId=myControlName; value=myValue}
... more shinyTexCode ...
\end{shinyConditionalText}
```

The text you enter for "myControlName" must exactly match the "inputId" of one of your input controls. The "more shinyTex code" is visible only when the specified control takes on the specified value. Do not use quotes and do assure that "myValue" matches one of the control's values exactly.

Internal Links: The shinyInternalLoc and shinyInternalLink commands

Two commands are needed to place a link that moves the user within the shinyTex app. The `\shinyInternalLoc{name=myLocation}` command is placed immediately before any section of shinyTex code that you want to allow the user to link to (jump to) by clicking on a link elsewhere. The `\shinyInternalLink{name=myLocation; text=my link text}` command is placed where you want the link with the "my link text" to appear. Note: internal links do not work across tabs, only within the same tab (or anywhere if tabs are not being used). The internal link color and background color are controlled by the "defs" statements in the Latex preamble of the document (see "Starting a new app", below).

A button to switch tabs: The shinyGotoTab command

The command to create a button that allows the user to jump to a different tab is `\shinyGotoTab{name=myTabName; text=my jump text}`. The "`myTabName`" must be the name of a tab within your app, and must be spelled exactly correctly, including upper vs. lower case. Additionally you can use `textColor=`, `backgroundColor=`, and `borderColor=` to set the colors of the `gotoTab` button to one of the **standard CSS colors**. If you know about CSS styles, you can use style=, but because the semicolon is used to separate options in shinyTex, you must substitute the "|" character for the semicolon when specifying several CSS style elements.

Images: The shinyImage command

The command (not environment) to place an image in your app is `\shinyImage{filename=myFilename; width=#; height=#}`. Only the filename is required. Depending on the browser, support file formats include jpeg, gif, png, svg, and bmp. The width and height are given in pixels. The image file must be present in the "www" subfolder of the main folder. As usual, when entering the options, separate the options with a semicolon (";") and do not use quotes.

Audio clips: The shinyAudio command

The command (not environment) to place an audio clip in your app is `\shinyAudio{filename=myFilename; type=myMIMEtype; autoplay=TF)`. Only the filename is required. The default MIME type is "audio/mp3". Set "autoplay" to `TRUE` to have the audio start automatically when the app loads. The audio file must be present in the "www" subfolder of the main folder. Audio types other than the default may require the user to load the appropriate browser plug-in. As usual, when entering the options, separate the options with a semicolon (";") and do not use quotes.

Video clips: The shinyVideo command

The command (not environment) to place a video in your app is `\shinyVideo{filename=myFilename; type=myMIMEtype; width=#; height=#; autoplay=TF)`. Only the filename is required. The default MIME type is "video/mp4". The width and height are given in pixels. Set "autoplay" to `TRUE` to have the video start automatically when the app loads. The video file must be present in the "www" subfolder of the main folder. Video types other than the default may require the user to load the appropriate browser plug-in. As usual, when entering the options, separate the options with a semicolon (";") and do not use quotes.

Native Shiny UI Code: The shinyUI environment

If you want to use a Shiny feature that has not been incorporated into shinyTex yet, you can put native Shiny user interface code into a `shinyUI` environment, i.e., between `\begin{shinyUI}` and `\end{ShinyUI}`.

Native Shiny Server Code: The shinyServer environment

If you want to use a Shiny feature that has not been incorporated into shinyTex yet, you can put native Shiny server code into a `shinyServer` environment, i.e., between `\begin{shinyServer}` and `\end{ShinyServer}`.

Capturing and running user-entered R code: The shinyRFeedback and shinyRHiddenFeedback environments

The shinyRFeedback environment is used to capture and run user-entered R code and provide detailed feedback. (Currently the feedback computes before the user completes her entry unless you modify the system shiny.min.js file and comment out the `$(a).on("keyup.textInputBinding input.textInputBinding",function(a){b(!0)}),` text.) Note that there is some danger to this feature, because a user can alter files that your Shiny server gives that access to, including deletion of your shiny apps. `ShinyTex` environment will attempt to stop all but the most nefarious users from altering your files.

The environment creates both an "input box" where the user can type R code and a "output box" where feedback is provided based on a "recipe" that constitutes the contents of the `shinyRFeedback` environment.

The shinyRHiddenFeedback environment is an optional supplement to each shinyRFeedback environment which can be used to provide different, e.g., more detailed, feedback in an output box that is hidden until the user clicks a checkbox to open it.

Note that in addition to the input (a single line or multiple inputbox) and the output (a verbatimOutput that contains the feedback), the environment creates reactive code that manages running the user's input in R, and an RFeedback() function that analyzes the results of running the code to provide the specific feedback you desire.

The syntax of the RFeedback environment is:

```
\begin{shinyRFeedback}{inputId}{[option pair; [option pair; [… ]]]}
-- your recipe for feedback --
\end{shinyRFeedback}
```

The `inputId` is unquoted text that names the components created by this environment.[1]

The option pairs are of the form `name = value`, where the name is one of `nrows`, `ncols`, `label`, `feedbackLabel`, `submitLabel`, `allowSemicolons`, `vars`, `carryVarsOver`, `monitorAll` and `monitorErrors`, and the value is un unquoted string. As usual options are separated by a semicolon (;). None of the options are required, and `{}` may be used if none are entered. The `nrows=` option sets the number of rows for user input (default `1`, range `1` to `30`). The `ncols=` option sets the number of columns for the user input (default `30`). The `label=` option sets the text placed above the input box (default: `R code prompt(>)` if `nrows=1` or `R code input:` otherwise). The `feedbackLabel=` option labels the feedback box which is placed below the output box (default: `Feedback:`). The `submitLabel=` options has effect only if nrows>1, in which case it labels the button that the user uses to submit their multi-line R code (default: `Submit R code`). For a single line input, the default behavior is to prevent the user from executing multiple R statements by separating them with a semicolon ";". If you want to allow such behavior, use `allowSemicolons=TRUE`.

See below for details of `vars=` and `carryVarsOver=`.

Because writing feedback code is tricky, you may want to monitor the users' behaviors to check the relationship between their inputs and your corresponding feedback. Any monitored information is placed in a file called "ERRORLOG.txt". The default behavior is `monitorErrors=TRUE` and `monitorAll=FALSE`, in which case each time a user inputs something that your

[1] If you are a more technical user of `shinyTex`, it may interest you to know that, e.g., if you use `{R1}` to specify your `inputId`, then the system will create a `textInput` or `textareaInput` (depending on the `nrows=` option) called `readRCodeR1`, a `verbatimTextOutput` called `rCodeFeedbackOutR1`, and a reactive function called `readRCodeRtnR1()`.

code does not catch in an error-free way, a line is added to "ERRORLOG.txt" containing the date and time, the `inputId`, the line of user code that caused the error, and R's error message. This will not happen if you set `monitorErrors= FALSE`. If you want to record all user inputs for this specific `inputId`, you can set `monitorAll=TRUE`.

Overview of how feedback works:

The feedback for a given user input that is typed into the input box provided by the RFeedback environment is based on the simple structure explained here.

First the code is checked to see if it is empty. If it is, then either the default message (`(no code)`) appears in the output box, or your author-specific message is displayed.

Second, if the input code is not empty, then it is checked for attempts at nefarious behavior. If the code appears to be trying something nefarious, then it is not run, and either the default message (`Please do not attempt to use dangerous functions!`) or your author-specific message is displayed.

Third, an environment ("sandbox" workspace) is created within which the author can create variables accessible to the user's code and in which the user may create their own variables. Note that the feedback formula can examine the variables in the workspace and customize feedback based on, e.g., what variables the user created. The initial variables in the workspace are controlled by the `vars=` RFeedback option. The default is an empty workspace. To place variables in the workspace, the syntax is a comma-separated set of "name=value" pairs where the name is any valid R variable name and the value is any valid value. E.g., `\shinyRFeedback(R1}{nrows=4; vars= a=3, b=list(n=1:3, L=LETTERS)}` will create an input/output pair on the web page with the internal name "R1", with 4 rows for the user input, and with two variables (a numeric vector of length one called "a" and a list called "b") in the user's workspace.

The "sandbox" workspace comes in two varieties. If you use the default option `carryVarsOver=FALSE`, then each time the users enters something new in the box, their code is evaluated relative to the workspace defined by the vars= option. But if you choose `carryVarsOver=FALSE`, then any variable additions or deletions made by the user are carried over through repeat code submissions. E.g., if the default empty workspace is used and variable are set to carry over, then the user could enter `a=3`, submit that code, and the enter `print(a)`, and the code would run correctly.

Fourth, an attempt to run the user's code (in the "sandbox" workspace) is made. The results are captured in several variables, which may be useful for providing complex feedback. Simple feedback may not require you to explicitly use these captured variables, but they will be used "silently" by `shinyTex` to control the feedback.

The automatically created variables are

- INPUT contains all of the user's R code that they submitted.
- OUTPUT shows any normal R output that results from running the code. E.g., a print() statement has output, but an assignment statement does not.
- VARNAMES contains the names of the variables in the "sandbox" workspace after the users command(s) are executed.
- VARS is a named list containing the values of all of variables named in VARNAMES.
- ERROR is a logical variable that is set to TRUE if the user code caused an error. Note that with multi-line code (or multiple statements separated by semicolons), there may be valid output in the OUTPUT variable from code the comes before the erroneous code.
- ERRORCODE contains the specific line of code that caused an error.
- ERRORMESSAGE contains R's error message in response to the user's erroneous code.

Fifth, each of your author-defined "groups" of code are run. There are three types of groups. ERRORGROUPs are entered only if the user's code caused an error and the parenthesized condition, if any, is true. NOERRORGROUPs are entered only if the user's code did not cause an error and the parenthesized condition, if any, is true. Ordinary GROUPs are entered based on the conditions you specify (see below) or unconditionally if you do not include a condition.

Optionally, each group may start with a CALC section, which can be used to calculate any intermediate results that will make specification of conditions or feedbacks simpler.

After the optional CALC section, each group consists of an ordered set of further "section"s and "feedbacks". ***Within each group the user sees only the feedbacks associated with the first section whose condition is true.*** Each group may have a DEFAULT condition which contains the feedbacks you want the user to see if none of the section conditions in that group are met. If you do not specify a DEFAULT for a given group, then that group may produce no feedback if none of the conditions are met.

Sixth, if it turns out none of the groups produced any feedback, then the default (`> Sorry, the feedback algorithm failed. Try entering something else. <',`) or author-defined GLOBALDEFAULT feedback is shown to the user.

How to enter the feedback "formula":

The feedback recipe is entered using the following syntax. Indenting is optional, but encouraged. In this syntax, square brackets indicate optional information. The terms "feedbacks", "calculations", "SECTION", and "conditions" are explained below, and "+" means zero to many of the following elements are allowed:

```
\shinyRFeedback{inputId}[{feedbackOptions}]
[EMPTY:
  feedbacks]
[NEFARIOUS:
  feedbacks]
+[ERRORGROUP[(conditions)]:
  [CALC:
    calculations]
  +[SECTION(conditions):
     feedbacks]
  [DEFAULT:
     feedbacks]
]
+[NOERRORGROUP[(conditions)]:
  [CALC:
    calculations]
  +[SECTION(conditions):
    feedbacks]
  [DEFAULT:
    feedbacks]
]
+[GROUP[(conditions)]:
  [CALC:
    calculations]
  +[SECTION(conditions):
     feedbacks]
  [DEFAULT:
     feedbacks]
]
[GLOBALDEFAULT:
  feedbacks]
```

The valid section names are summarized here:

Section name	**Defines output when:**
OUTPUTHAS( ):	At least some code ran and the output matches the regular expression you specify.
OUTPUTLACKS( ):	At least some code ran and the output does not match the regular expression you specify.
INPUTHAS( ):	At least some code ran and the input matches the regular expression you specify.
INPUTLACKS( ):	At least some code ran and the input does not match the regular expression you specify.
ERRORCODEHAS( ):	An error occurred and the user's R code matches the regular expression you specify.
ERRORCODELACKS( ):	An error occurred and the user's R code matches the regular expression you specify.
ERRORMESSAGEHAS( ):	An error occurred and R's error message matches the regular expression you specify.
ERRORMESSAGELACKS( ):	An error occurred and R's error message matches the regular expression you specify.
VARCLASSIS( ):	Using (var, class), the variable you specify exists and its class is the one you specified.
VARCLASSISNT( ):	Using (var, class) the variable you specify exists and its class is not the one you specified.
ENVHAS( ):	The environment contains the specified variable.
ENVLACKS( ):	The environment does not contain the specified variable.
ENVNOTEMPTY:	The environment is not empty, i.e., the user created at least one variable.
ENVEMPTY:	The environment is empty.
CONDITION():	Any defined condition including complex combinations of the other conditions is true.

You can use none or as many of these feedback sections in each group as you need to define your custom feedback. As shown above, you can also optionally end any group with a DEFAULT:. (Some combinations are useless, such as repeats of ENVEMPTY and ENVNOTEMPTY, as well as ENVHAS after ENVNOTEMPTY and ENVLACKS after ENVEMPTY.)

Note that some sections names take parenthesized options and others do not. Also, the VARCLASS sections need two comma-separated inputs inside their parentheses. For all of the eight section names that take regular expressions, you can add an asterisk between the section name and the left parenthesis to suppress regular expression matching (i.e., to use R's fixed=TRUE option). **If you are not proficient at regular expressions, you should use the INPUTHAS*(), etc., form of these eight section names.**

The line(s) after each section name ("feedbacks" in the syntax summary above) are used to generate the text the user sees inside the "feedback box" if the condition is triggered. The simplest cases are literal text (no complex expressions needed), in which case the text should be placed within single (') or double quotation marks ("). Alternatively, you can use R code that constructs the feedback text, possibly using the special pre-defined variables such as `VARNAMES` as described above.

Here is a simple example:

```
Enter any R code to see what it does.  You can repeat the process as
often as you like.
\begin{shinyRFeedback}{anyCode}{prompt=Simulated R prompt (>),
                                ncols=40}
ERRORGROUP:
  DEFAULT:
  "That is not valid R code."
GROUP:
  DEFAULT:
  "Your code produced the following output:"
  cat(OUTPUT, collapse = "\n")
\end{shinyRFeedback}
```

This will produce a single line input box and an auto-sizing feedback box. Before the user enters input or if they erase all input, they will see the default EMPTY text, which is `(no input)`. If the user enters text into the input box that is invalid R code, they will see `That is not valid R code`. If they try something like file.remove("ui.R"), they will see the default "NEFARIOUS" text. Otherwise they will see the output normally shown after running their command, because `OUTPUT` is predefined to be the user's R output.

The parenthesized condition for the [[NO]ERROR]GROUP](): lines or the general CONDITION(): lines can be quite general and may include parentheses, "!" for "not", "&&" for "and", "||" for "or", any of the other predefined variables such as OUTPUT, any CALC section variables, and any of the predefined conditions such as "VARCLASSIS(var, class)".

Here is another example, which uses a CALC section to detect that an error was caused by the user trying to assign something to a reserved variable name:

```
ERRORGROUP:
  CALC:
    eqRE = regexpr("[a-zA-Z_.]+[[:space:]]*=", INPUT)
    if (eqRE == -1) {
      reserved = FALSE
```

```
  } else {
    nchRE = attr(eqRE, "match.length") - 1
    varName = str_trim(substring(INPUT, 1, nchRE))
    reserved = nchar(varName) != nchar(make.names(varName))
  }
CONDITION(reserved):
  cat("You used a reserved variable name ('", varName, "').", sep="")
  "Try a different name."
DEFAULT:
  "Your code produced an error.  The error message is:"
  cat(ERRORMESSAGE, sep="\n")
```

When writing feedback, be careful of a few complexities. First, because `shinyTex` is based on LaTeX syntax, the percent sign (%) mean "the rest of the line is a comment." But, R has several operators that include a percent sign. So, e.g., to perform modulo arithmetic, use if `(x\%\%y==0)` instead of `(x%%y==0)`. Also, there are places where quotation marks also need to be preserved using leading backslashes, e.g., `RFeedback{myFeedback}{env= a=\"string\"}`.

In addition to the predefined variables such as "VARNAMES", any CONDITION() statement or CALC block may use "backtick" notation to refer to any user created variables. E.g.,

```
CALC:
   xIn1to5 = "x" \%in\% VARNAMES && is.numeric(`x`) &&
             all(`x` \%in\% 1:5)
```

Note that conditions like VARCLASSIS() automatically also check for the existence of the variable, and thus do not cause an error if the variable does not exist. This special feature does not apply to backtick variables, so additional code as in the above example may be needed. In this example we first check that the user created variable *x*. Note that VARNAMES is a character vectors, so in that case we use `"x"`. But then we assure that it is numeric using backtick notation. We use backticks around the variable because in ordinary R code we need a variable name, not a quoted variable name inside the `is.numeric()` parentheses, but if you just put *x*, that would presumable refer to a variable x that you (the author) created, not an *x* variable created by the user of the app. The backticks tell `shinyTex` that you are referring to a user created variable.

The shinyRHiddenFeedback environment

Sometimes it is worth showing some aspects of feedback only when the user desires, e.g., either as optional additional feedback or when you want the user to guess what the result will be before seeing it. In that case you can use the `\shinyRHiddenFeedback` environment. The `inputId` for this environment must match the `inputId` of a prior `\shinyRFeedback` environment. The only options available are `openLabel`, `monitorError` and `monitorAll`. The `openLabel=` option defines the text on the check box that opens the output box (default: `View detailed`

`feedback`). The monitor options are the same as for `/shinyRFeedback`. The feedback formula inside the environment follows all of the same rules as for `/shinyRFeedback`.

"Checks" of your RFeedback and hiddenRFeedback code

Writing good feedback code for user R input is challenging. A cycle of altering the feedback code, compiling and running the app, finding the input box in the app, and trying one or more examples of user input to see what the corresponding feedback output look like can be tedious. The "feedback checks" are designed to simplify this process.

Remember that each RFeedback (and possibly its corresponding RHiddenFeedback) has an id. In this section we will use "Foo" as an example feedback id. To check what feedback "Foo" might give for different inputs, create a plain text file called "FooCheck.txt". In the file enter one or more lines corresponding to some user input, then enter a line of dashes (at least three) to separate one set of user input from the next, and repeat as desired.

You can create these check (input) files for whichever RFeedbacks you like: none or for a few or for all of the RFeedbacks.

When you compile your app with `shinyTex()`, a check output file called FooCheckOut.txt will be produced in your working directory. This file will show each set of input code lines, the corresponding feedback output, and, if there is a corresponding RHiddenFeedback, the hidden feedback (delimited by sets of left and right parentheses to indicate the "hidden" nature of the feedback, i.e., that it will not be initially visible in the app.

Once you are satisfied with your feedback code for a particular RFeedback environment, it is recommended that you rename "FooCheckOut.txt" to "FooCheckOutOK.txt" to indicate that this is output that you are satisfied with, and which you may want to re-verify in the future. If "FooCheckOutOK.txt" exists, then whenever you next run `shinyTex()`, the "FooCheckOut.txt" file you create will be compared to the "FooCheckOutOK.txt" file and a warning will be given if they differ.

As a further convenience, particularly for a larger, more complex app, you can place the "FooCheck.txt" and "FooCheckOutOK.txt" files in a subdirectory, e.g., called "check". Then `shinyTex` will not bother wasting time checking the inputs unless you use the alternate compilation call, `shinyTex(checkDir="check")`.

In summary, "FooCheck.txt" is an input file of example user inputs and may be placed in the working directory or a named subdirectory, "FooCheckOut.txt" is created in the same location as "FooCheck.txt" and contains the feedback "output", and "FooCheckOutOK.txt" is "frozen" output against which "FooCheckOut.txt" is compared. As each RFeedback environment is encountered by `shinyTex`, first an input file is searched for in the working directory, and then if one is not found a new search is made in the "checkDir" directory, if specified. If an input file is found, an output file is produced in the same location containing output for both the main

RFeedback and possibly and corresponding hiddenRFeedback. Finally, if the directory containing the input and output files also has a "frozen (OK)" file, the new output and the frozen output are compared.

Stored (Reusable) Feedback Code: The shinyStoreCode environment

If you find yourself using some of the same lines of R Feedback Code repeatedly, you can place the code in a `shinyStoreCode` environment, and then used the `\shinyInsertCode{}` command inside multiple `shinyRFeedback` environments to insert the same code into several `shinyRFeedback` environments.

Here is an example:

```
\begin{shinyStoredCode}{showEnviron}
  NOERRORGROUP:
    ENVEMPTY:
      'Your environment is empty.'
    ENVNOTEMPTY:
      cat('Your environment has', length(VARNAMES), 'variables.')
\end{shinyStoredCode}
```

and then elsewhere (usually at least two times)

```
\begin{shinyRFeedback}{createVar}
  ERRORGROUP:
   'Your code caused an error.'

  \shinyInsertCode{showEnviron}
\end{shinyRFeedback}
```

The RFeedback environment will function the same as if the "showEnviron" code had actually been included in the feedback "formula".

Development cycle and debugging

Setup

To **install** the latest version of `shinyTex` on your computer, download the zip (Windows) or tar.gz (Mac) file from http://www.stat.cmu.edu/~hseltman/shinyTex/. Then install the package in R. E.g., in RStudio, click "Packages", and then "Install", change "Install from" to "Package Archive File", then click "Browse…" and navigate to the downloaded file, and finally click "Open". This needs to be done only once. (If a new version of `shinyTex` is released, repeat the install procedure in any folder, but before running the `library(shinyTex)` command.)

Planning the app

Although a single page app is possible, most TEL apps use multiple tabs to present different parts of the "lesson". Possible/suggested tabs include

i. Pre-test (Am I ready to use this app?)

ii. Context and/or background and/or goals
iii. Interactive exploration of simulated data
iv. Alternate real data sets
v. Coding in one or more computer languages / statistics programs
vi. Post-test
vii. Further links

ShinyTex does not support highly controlled guidance through the TEL experience. It is designed for motivated learners who will control what they read or interact with and when. Suggestions for sequential (or non-sequential) use of tabs can be placed at the top of the first tab, since all users will see that first. Hiding information for a later reveal is possible, e.g., through use of a "See solutions" check box (`\shinyCheckbox`) coupled with a conditional text section (`\begin{shinyConditionalPanel} ... \end{shinyConditionalPanel}`). And of course, you may place web links to other shinyTex apps or any other web resource freely throughout your app using `\href{}{} or \shinyLinkNew{}{}`.

Starting a new app

To **start a new app or collection of apps**, begin in a new project in RStudio (or a new folder in R). Run `library(shinyTex)` and `setupShinyTex()` at the R command prompt. If a new version of `shinyTex` is released, after installing it in an R session in any folder, you should rerun `setupShinyTex()` in each folder (though this may be unnecessary, depending on the types of changes incorporated into the new version).

For each R `shinyTex` session you will need to run `library(shinyTex)`. (You can avoid this in Windows if you create `.First=function()library(shinyTex)` and save your workspace.)

You can either put several apps in the same directory or each app in a separate directory. Each app is constructed by entering `shinyTex` code in a .tex file. If you have several apps in the same directory, then only one is "active" at a time. This means that when you process your code using `shinyTex()`, you can either enter the .tex file name as the argument or you can leave the argument off, and the most recently edited .tex file will be processed. It also means that when you run an app with shiny's `runApp()` command, the most recently processed app will be run and there is no way to run any other without re-running it through `shinyTex()`. But if you are deploying your apps through a shiny server, it may be convenient to keep several related apps in one directory on your computer, and copy the app files for the app you are working on to the server when you have it working the way you want.

For each app, begin by creating a new text file in R with a ".tex" extension or by using a copy of shinyTemplate.tex renamed to match your app.

You can add as many comment lines as you like, beginning each with "%". Comments can also go on a line after other non-comment code.

Enter any of the following statements to define the way the app title is produced. (Some code comments in the distributed app are also affected by these statements.)

```
\def\author{YOUR NAME GOES HERE}
\def\title{YOUR APP TITLE GOES HERE}
\def\titleFontPercent{SET RELATIVE HEIGHT; DEFAULT IS 100}
\def\titleColor{YOU CAN USE THIS LINE TO CHANGE THE TITLE COLOR}
\def\titleFontWeight{DEFAULT IS NORMAL; CAN BE 'bold'}
\def\titleFont{DEFAULT IS 'serif'}
\def\theme{A 'shinythemes' THEME NAME GOES HERE}
\def\backgroundColor{A COLOR FOR THE APP BACKGROUND}
\def\verbatimColor{A COLOR FOR VERBATIM ENVIRONMENT BACKGROUND}
\def\headerColor{A COLOR FOR THE BAKCGORUND OF THE APP HEADER}
\def\titleBackground{A COLOR FOR THE BACKGROUND OF THE APP TITLE}
\def\navbarBackground{A COLOR FOR THE TABS (NAVBAR) BACKGROUND}
\def\internalLinkColor{A COLOR FOR THE TEXT OF INTERNAL LINKS}
\def\internalLinkBackgroundColor{A BACKGROUND COLOR FOR INTERNAL
                                 LINKS}
\def\internalLinkFont{A FONT FOR THE TEXT OF INTERNAL LINKS}
\def\codeColor{A COLOR FOR THE TEXT OF \code{} text}
\def\codeBackgroundColor{A BACKGROUND COLOR FOR \code{} text}
\def\codeFont{A FONT FOR THE TEXT OF \code{} text}
```

The default is red text for the title. The color, font weight, and font "defs" are based on HTML standards for color, font-weight, and font-family. You can change the overall look of your app using shinythemes. Colors can be specified as the CSS color name (not case sensitive), or using the CSS #nnnnnn format. Any color, background color, font, or font-weight may be omitted or set to "internal" for the default value.

Next, place a `\begin{document}` statement after the "defs" and place an `\end{document}` statement at the end of the file.

Finally, you need to setup the basic structure of the app. If you want to use a structure other than tabbed pages, see above under "Overall structure" under "Details of creating the LaTeX authoring file". To make tabs, add one or more of the following between the "begin" and "end" document statements, being sure to add the brief tab names inside the curly braces (spaces and punctuation are allowed).

```
\begin{shinyTab}{your tab name goes here}
\end{shinyTab}
```

You will probably want to include some comments for each tab.

In general, the order that you place your text, environments, and commands within the shinyTab environment mirrors the way your users will see the app.

Entering the app content

Between each "begin" and "end" shinyTab statement pair, you will enter your app content. This consists of as many of the following as you like in any order:

i. Ordinary paragraphs of text, including those marked up with bolding, formulas, etc. (See "Text outside of inner environments" above.)

ii. Input controls, such as slider, checkboxes, etc. to receive user input. (See "Input controls" above.)

iii. R output, including plots (see the `shinyPlot` environment, above) and output from any R command (see the `shinyCodeOutput` environment, above). To support these two outputs, you may need to use reactive R code inside of a `shinyReactive` environment (see "Working with R code in a shinyTex app" above) or ordinary R code not specifically related to output using the `shinyCode` environment. These latter two may be placed near the corresponding output environments for convenience, but because they have no direct effect on what the user sees, their exact positions in your LaTeX authoring file is not important.

iv. Quizzes are positioned using the `\shinyQuiz` statement (see "Quiz Banks" below).

It is also possible to divide an entire tab or some vertical section of it into vertical sections ("columns"). Within each column you can enter the same types of information as in the list above. To make columns use this syntax:

```
\begin{shinyColumn}{7}
… lines of shinyTex code …
\shinyNextColumn{5}
… lines of shinyTex code …
\end{shinyColumn}
```

This example is for a two column section of a tab with 7 and 5 as the relative widths of the two columns. Add more `\shinyNextColumn` commands to obtain additional columns. The main rule, taken from Shiny, is that the relative widths across all of your columns should add to 12.

Another possibility is to have some portion of the app visible only conditionally, typically tied to either a `\shinyCheckbox` or a `\shinyRadioButtons` control. An example using a check box would be "solutions" made visible only after the user click on a "Reveal solutions" checkbox. An example using radio buttons is a set of data analyses on several different datasets chosen by the radio buttons.

The compile / test / edit / extend cycle

Once you have entered some shinyTex code in one or more tabs, you are ready to **"compile" your app**, i.e., you will use the `shinyTex()` R function to convert your LaTeX authoring file into a stand-alone Shiny app in the form of the files ui.R, server.R, several .html files, possibly some feedbackFor and/or hiddenFeedbackFor files, and possibly some Quiz.RData files. To compile your

authoring file, run `shinyTex()` or `shinyTex("myAuthoringFile.tex")`. If you have more than one app in the directory and you do not use the ".tex" extension, the most recently edited .tex file will be compiled.

Once you run the compile command, if you get back the ">" command prompt with a "**successfully created Shiny files**" message, you will know that your app compiled successfully. If you get just warnings, you should read them, and then decide if you want to respond to or ignore the warnings. If you get an **error** message, it will give you specific information (often with the line number of the code that is causing the problem) explaining how your code does not follow the shinyTex rules, and you must correct your code and recompile before proceeding.

Next you will want to run your app to test it. To **run the app**, you enter `runApp()` at the R/RStudio console. By default, the app first opens in the "RStudio viewer window" for quick viewing with some missing features; you will want to click "Open in browser" to see all of the features.

To **change the default for where an app opens**, open an R file, such as shinyTex.R and notice the small rectangular icon to the right of "Source" in the window header. Click the icon and change from "Run in Window" to "Run External".

When you run your app three things might happen. First, you might get a "run time" error, which is any error that `shinyTex` cannot catch, but which prevents Shiny from running your app. These error messages are generated by Shiny and/or R, and might not be very specific. Generally, this type of error relates to errors in R code you write in `shinyTex` environments, e.g., the `shinyReactive` environment or the `shinyPlot` environment or the `shinyRFeedback` environment. The `shinyTex` program can only catch some errors of this type, and errors appearing when you run your app are the ones `shinyTex` cannot detect. See "Troubleshooting" below for possible help. When you see error messages in the R console, you will want to close the web page and click "Stop" in the R console to get the R command prompt. Then you can correct the errors in your authoring file. If you have `shinyRFeedback` or `shinyRHiddenFeedback` environments in your code, and if they are the source of the error, you will get more information about the error by examining the last line of the AUTHORERROR.txt file in your directory. Remember to recompile before running your app again. Also, check AUTHORERROR.txt if you are entering R code into a feedback box in the app, and you see the message "> Sorry, the feedback algorithm failed. Try entering something else. <", which is an indicator that the user entered R code for which the author provided no feedback at all.

Second, your app might run but activating a particular tab or setting certain control inputs might cause an error message. Close the app, fix the errors in the authoring file, recompile, and re-run the app.

Finally, and best of all, when you run your app it may run fine.

Work with the app and note mistakes and areas for improvement or extensions. To make changes to the app, close the app's web page, click "Stop" in the R console, and then repeat the edit/compile/run cycle.

Debugging your app

If you cannot determine the cause of a particular problem with your app, you may want to use the R debugger, specifically the `browser()` function as discussed in Debugging in R. Shiny allows you to place one or more `browser()` commands in your code, and shinyTex allows you to do this for any `shinyReactive`, `shinyCode`, `shinyPlot`, or `shinyCodeOutput` environment. Remember to save your authoring file after adding the command(s) and re-compile before running the app. You are allowed to use syntax such as `if (x>5) browser()` to trigger the browser only under certain conditions.

Once the code runs up to the point where you placed the browser command (which may only happen after certain user inputs), you will see "browser [1]>" which indicates that you are running in R debug mode. Here you may examine or change the value of any variable, try out alternative commands, and/or step through your program. See any web page on debugging in R for details. You can examine the current variables using RStudio's "Environment" tab. Note that inside of Shiny code, a command like `print(x[[2]])` surprisingly does not have any effect. The workaround is to enter something like `junk = x[[2]]`, and then examine the value of "junk" in the "Environment" tab. Once you have finished debugging, enter "c" to complete running the code, then quit the app, and make any necessary changes to the authoring file.

If you are having difficulty debugging some RFeedback, you can use this trick. Open the offending FeedbackFor.R file and add a `browser()` command. The go directly to `runApp()` without using `shinyTex()` (which would overwrite your browser command). Then you will be able to step through the feedback processing, examining variables using the `junk=` trick above. When you do finally correct your code, be sure to do so in the original .txt file, and not the FeedbackFor.R file, or you will lose your corrrections.

Quiz Banks

Quiz Banks are a feature of shinyTex used to implement multiple choice (or true/false) quizzes. A quiz bank is a set of quiz questions. Each question consists of the text of the question, and for each answer, the answer text as well as "response" text that explains something about that answer (whether correct or incorrect). The test banks do not keep track of correct answers. You do have the option to randomize the order of presentation of the questions, as well as the order of presentation of the answers for each question. Particular questions can also be marked to always have a fixed answer order.

Creating app quizzes is a three step process. The steps are 1) creation of the quiz bank LaTeX file (a kind of auxiliary authoring file), 2) compilation of the quiz bank LaTeX file, and 3) adding a `\shinyQuiz` statement in your main authoring file at the place you want the quiz to appear.

Each quiz bank is created as a separate plain text LaTeX authoring file with a ".tex" extension. One useful convention for the file name is to use the quiz bank name (the one in the "myBank" position of the `\begin{quizBank}{myBank}` command) plus ".tex". If you use this convention, then if you forget to compile a quiz bank or re-compile it after making a change, shinyTex will be able to compile it automatically.

There may be several different quizzes per app, each appearing at a different place in the app. Each file starts with `\begin{shinyQuizBank}{myBankName}` and ends with `\end{shinyQuizBank}`. (The `\begin{document}` and `\end{document}` commands are not used in a quiz bank.) The quiz bank name must be a single word that follows the rules of R variable naming. Only one quiz bank is allowed per quiz bank file, but each app may have as many quiz bank files as desired. (It is OK to use the same quiz bank in more than one app; you only need to have the myBankQuiz.RData file in your current directory to use a quiz.)

Comments are allowed in quiz banks. Any text beginning with a percent sign ("%") through the end of the line is a comment.

Between the begin and end `shinyQuizBank` lines, you can put as many quiz questions as you like. The format for quiz bank multiple choice quiz questions is:

```
\begin{shinyMCQ}{question text}[{fixed}]
  \shinyMCAR{answer text}{response text}
  \shinyMCAR{answer text}{response text}
   ...
\end{shinyMCQ}
```

Each question must have at least two choices, and the maximum number of choices is not limited. The question text will appear in the app as well as all of the answer texts. The app shows an open circle to the left of each choice's answer text, and when the user clicks inside that circle, the corresponding response text is revealed (until another choice is made). The code following the question text, `{fixed}`, is optional, as indicated by the square brackets; if it is included, then the answers for that particular question will never be randomized.

To include a math formula in a quiz question, answer, or response, use `$...$` for in-line math or `$$...$$` for offset math. Because all three of these quiz components are inside of curly braces, to use a curly brace as part of a math formula in a quiz, use "`\{`" instead of "`}`" and "`\{`" instead of "`}`". Currently other enhancements, such as `\mathbf` are not supported inside of quizzes.

After creating the quiz bank LaTeX file, or whenever you make changes to it, you need to compile (or-recompile) that file using the R function

`compileQuizBank("myBank")`, where ".tex" is implied. When the quiz bank is compiled, a myBankQuiz.RData file is created. This is a binary (non-human-readable) file in a format suitable for run-time creation of quizzes. If you forget to compile or re-compile your quiz bank, shinyTex will do so automatically if the name of the ".tex" file matches the name of the quiz bank it contains.

To place a quiz inside of an app, use the shinyTex command `\shinyQuiz{bank=myBank; addNumbers=TF; randomizeQuestions=TF randomizeAnswers=TF; max=#}`. Only the quiz bank name, "myBank", is required. TF indicates that you must enter `TRUE` or `FALSE`. "#" indicates a number. By default, 'addNumbers' is set to `FALSE` indicating that the quiz questions are not numbered, 'randomizeQuestions' is set to `TRUE` indicating that the quiz questions are presented in a random order rather than in the order listed in the quiz bank file, and 'randomizeAnswers' is set to `TRUE` indicating that the order of the answers for any question not marked as "fixed" in the quiz bank file will be randomized. Use "max=" if you want to limit the number of questions the user will see; excess questions are dropped randomly.

Examples

Chi Square:	Run the app	Authoring file	(quiz file) (quiz file)
Y as X:	Run the app	Authoring file	(quiz file) (quiz file)

Troubleshooting

It is recommended that you use appropriate vertical white space (blank lines) between all independent elements of your authoring code to make spotting problems easier. Also, especially when you are new to shinyTex, test your app frequently rather than writing a large amount of code before testing.

The most common errors are mismatched curly braces and mismatched `\begin{someEnvironment}{myOptions}` and `\end{someEnvironment}` statements, so look for those first. Also remember that shinyTex is case sensitive, so, e.g., `\begin{shinyQuizBank}` does not match `\end{shinyQuizbank}`.

Here are some error messages and suggestions for correcting the problem:

- "Error in shinyAppDir(x) : **App dir must contain either app.R or server.R**." This indicates that you have not yet run `shinyTex()` on your authoring file.
- "Error in source … ui.R:24:39: **unexpected symbol**" This commonly is due including quotation marks in the options for the input control commands.
- "Warning messages: …: **unexpected Shiny command(s)/ environment(s)**: \begin{shinyFoo} \shinyBar \end{shinyFoo}" This indicates use of commands or environments that are not defined in shinyTex, often just a simple misspelling. For this example, an environment called "shinyFoo" was

used, but that is not a valid shinyTex environment. Also the command called `\shinyBar` was used, but that is not one of the valid shinyTex commands.

- "Warning messages" In processText(paste(tex[typeBegin[block]: typeEnd[block]], collapse = " "), @114-114 **un-handled LaTeX commands**: \bf" The deprecated `\bf{}` was used where `\textbf{}` is correct. The error is on the specified line (line 114 in this example).
- "Warning in … **incomplete final line** found on '…'" You did not press the Enter/Return key after typing the final line in the specified file. Just go back and do that now to remove the warning.
- "Warning in … : **Note: compiling ….tex**" You forgot to run compileTestBank() before running your app, so your test bank was automatically compiled.
- "Warning in … **Note: re-compiling modified ….tex**" shinyTex detected that you have made changes to your test bank but did not recompile it, so it was automatically re-compiled.
- "Error: **could not find function 'runApp'**" You forgot to load the shiny library; type `library(shiny)`. Error: could not find function "shinyTex"
- "Error: **could not find function 'shinyTex'**" You forgot to load the shiny library; type `library(shiny)`.

LaTeX primer

The LaTeX typesetting language describes a document in a formalized way. The LaTeX "code" is plain text that you write, which is normally converted to a viewable document using the latex command on your computer system (or some other way if LaTeX is run within a particular Graphical User Interface (GUI). The shinyTex system takes a special form of Latex and uses the `shinyTex()` command in R to create all of the files that constitute a complete Shiny app.

LaTeX uses the percent sign "%" to indicate comments. Everything on any line starting with `%` and up to the end of that line is a comment.

LaTeX uses a backslash character (`\`) to indicate its special features. The three basic forms are 1) an isolated backslash to make something special happen to the next character, e.g., `\%` is used to put a percent sign into a document, i.e., to override the commenting function of the sign; 2) short term effects in the form of a LaTeX "command" using syntax like `\someCommand{text affected by the command}`; and 3) environments, which are extended sections of text, beginning with `\begin{someEnvironment}{myOptions}` and ending with `\end{someEnvironment}` and are affected by whatever the nature of the environment is.

A LaTeX document begins with `\documentclass[myOptions]{someClassName}` to indicate the kind of document being produced. This is not needed for shinyTex. Several complex optional command are available in LaTeX to control the way the document is produced, but the only ones pertinent to shinyTex are the specific "\def" commands described above under "Starting a new app".

The bulk of any LaTeX document is placed within a "document" environment, i.e., between the line `\begin{document}` and `\end{document}`.

LaTeX is paragraph oriented and generally makes no distinction between a single space and several spaces, and makes no distinction between a single blank line and multiple blank lines. Several lines together constitute a paragraph, and the locations of the line breaks within a paragraph are immaterial. Generally, environments work like separate paragraphs so blank line before or after the environment may be optional.

In shinyTex, some commands (input control statements and $$myFormula$$) and all environments work like separate paragraphs. Although it is good form to separate each of these from the rest of your authoring code using blank lines, it is not necessary.

Formulas in LaTeX are complex and powerful. Full details may be found in the LaTeX Wikibook and elsewhere. Here is a brief overview. Put your formulas inside of single dollar signs (`$`) for in-line formulas and double dollar signs if the formula should be offset as its own paragraph. Use `\alpha, \Alpha, \beta, \Beta`, etc. for Greek letters. Use `\log, \sin`, etc. for standard function names (to assure the correct font). Use `\sqrt{myExpression}` for a square root. Use `\frac{myNumerator}{myDenominator}` for fractions. Use a carat (`^`) for superscripts and an underscore (`_`) for subscripts. If more that an single character is to be raised or lowered, but all or the characters inside of curly braces, e.g. `x_b^{2y}` to get x_b^{2y}. To make a summation, you can use something like `\sum_{x=1}^{\max(y)}z^i` to get $\sum_{i=1}^{\max(y)} z^i$. See the documentation for the many additional features.

Frequently asked questions (FAQ)

- **Why not just author in Shiny?** Shiny authoring requires maintaining two coordinated files the server.R and the ui.R files, which is considerably more complex than the shinyTex system. Also, working in Shiny requires a greater level of knowledge about HTML and JavaScript, at least for using more advanced features. In addition, reading shinyTex authoring code is much easier that the ui.R and server.R files, both for someone who knows Shiny/shinyTex and someone who does not. Specifically, there will be far fewer quotes, backslashes and commas in the authoring text. As an example what must be written in Shiny as "\\(\\sigma\\)" is written "σ" in shinyTex.
- **What files are created by shinyTex?** When you run the R command `shinyTex("myApp")`, the files produced are ui.R and server.R. In addition, a number of ##-##.html files are produced. Old versions of these files are deleted at the start of the process. Also, when you run `compileQuizBank("myQuizBank")`, a file called "myBankQuiz.RData" is

produced where "myBank" is replaced by the bank name given in the `\begin{shinyQuizBank}{myBank}` command.

- **What files are needed to deploy a shinyTex app?** In addition to the ui.R, server.R, the #-#.html files, any feedbackFor*.R, any hiddenFeedbackFor*.R files, and the *Quiz.RData files produced by shinyTex, your users need access to shinyTexAux.R and www/shinyTex.css . Users do not need access to any of your .tex authoring files nor shinyTex.R. If you have any audios, videos or other media files, they are also required, and should be placed in in the www subfolder. Users' systems must have the R libraries "stringr" and "shiny" installed, but not "shinyTex". In this context "have access" means have the files on their system if they are running Shiny in R directly, or that these files are in a zip file available on the World Wide Web if users are running Shiny on their systems but using the shiny `runUrl()` function, or if users are running the app directly from a browser using your server, then these file must be on the server.

Minimal setup for users when a Shiny server is not used

If you do not use a Shiny server to deploy your app, your users will need to download and install R from r-project.org. Running R under RStudio is suggested, but not required. It is required to install the "shiny" and "stringr" packages in R or RStudio. Finally shinyTexAux.R must be in the working directory with all of the app files (ui.R, server.R, and all *.html and *Quiz.RData" files), and shinyTex.css must be in a subdirectory called "www". If you have video or other media files, they must be placed in the "www" subdirectory, too.

Unless you use a .First function, to run an app the user must type:

```
library("shiny")
runApp()
```

Implementation details

ShinyTex creates both ui.R and server.R files as required by Shiny. Each independent section of the app that can be constructed using only HTML (but not any special shiny features) is stored in a file named #-#.html, and that file is incorporated into the app using `\includeHTML()` in the ui.R file. Wherever possible, e.g., for plots, shinyTex invents names to link the ui.R file and server.R file.

Quizzes are implemented by storing the Shiny tag.list objects for the conditional panel set for each question in a list containing all questions. This list is stored in an R "save" object and stored in "myBankQuiz.RData" when the quiz is compiled. At run time, the `quiz()` function can optionally randomize the questions, add

questions numbers, and randomize the answers. Then the final version of the quiz is passed to Shiny for incorporation into the current app.

Glossary

app: an interactive application created by Shiny tex that can be run in a web browser using Shiny

authoring: specifically, the process of entering text and code into a .tex file using shinyTex conventions; generally, the whole processes of creating a .tex authoring file and testing and revising it

command: shinyTex code beginning \shiny and designed to implement a specific shinyTex feature into an app (contrast with environment)

deployment: the process of making your app available to others

directory: same as folder; a feature of your operating system that is a container for files (and other directories)

dvi: Device Independent file format; a file format created by LaTeX

environment: a multi-line LaTeX or shinyTex code segment with a specific function; all shinyTex environments begin with a line like `\begin{shinyFoo}{options}` and end with a line like `\end{shinyFoo}`

feedback code: Specialized code in a `shinyRFeedback` or `shinyRHiddenFeedback` environment used to define details of feedback on user-defined R code.

folder: see directory

horizontal rule: an HTML feature that draws a line across the page

HTML: Hypertext Markup Language; the basic language understood by all web browsers upon which Shiny and ShinyTex are based

LaTeX: A typesetting system used by mathematicians and statisticians

markup: code that indicates that certain text should be represented in a special way

pdf: Portable Document Format: a platform-independent file format supported by Adobe Reader and others that represents documents

project: a feature of RStudio that keeps maintains separate work areas for separate endeavors

R: a free open-source statistical computing system; Shiny and shinyTex are built upon R

RStudio: a free Graphical User Interface (GUI) that makes working with R easier and more efficient

Shiny: a free system developed by the RStudio organization (but not requiring the RStudio GUI) that provides web apps based on R

shinyTex: the free authoring system built on Shiny which is the subject of this document

sidebar: a vertical section of an app with a separate function from the other (main) vertical section

sourcing: the process of loading R code into R; it is accomplished with the `source()` command or the menus in R.

subdirectory: a directory that is contained inside another directory; in this document it refers to a directory inside the main directory containing the code for an app

subfolder: see subdirectory

tab: an HTML mechanism for simulating stacked pages, only one of which is visible at any time

TEL: technology enhanced learned (computer aided learning)

TeX: the basic typesetting language up which LaTeX is built

URL: Uniform Resource Locator; a web "address"

working directory/folder: the directory holding the main files for an app