Developing Interactive R Problem Sets with RTutor

Sebastian Kranz, Ulm University

Version from 2015-05-26

Contents

1 Brief overview

1.1 RTutor

RTutor is an R package that allows to develop interactive R exercises. The interactive exercises directly test a student's solution and provide hints if not everything is correct. Unlike purely web-based approaches, as e.g. https://www.datacamp.com/, RTutor allows off-line or on-line use. Problem sets can either be solved as Markdown .rmd file directly in RStudio or in a browser based interface powered by Shiny. I have already used the markdown based RTutor problem sets in my Master courses at Ulm University, with quite positive feedback. Furthermore, I have supervised several students who have created very nice web-based RTutor problem sets that allow to replicate the analyses of well published economic articles in an interactive fashion.

Examples and additional material can be found on RTutor's Github page:

https://github.com/skranz/RTutor

This document gives you an overview how to generate an interactive RTutor problem set. There is an extra document that gives additional advice for creating an RTutor problem set as part of a Bachelor or Master thesis.

1.2 Install the newest version of RTutor

Installation instructions are given on RTutor's Github page:

https://github.com/skranz/RTutor

1.3 Create an R Markdown solution file

You create an interactive problem set by writing a solution file in R markdown format. In addition to the exercise description and a sample solution, the solution can contain manually adapted test, hints, or other commands like giving awards for a correct solution. Where no manual hints or tests are specified, RTutor will automatically generate tests and hints.

Here is the code from the example solution file "Example_sol.Rmd" that you find in the problem sets directory of the RTutor library. We will explain its structure in detail in the chapter on solution files.

```
# Problemset Example

Example of an RTutor interactive Problemset
Author:  Sebastian Kranz
Date:    12.06.2014
```

````
Adapt the working directory below and press Ctrl-Alt-R (run all chunks).
This creates the problem set files and the sample solution from this solution file.

#< ignore
```{r " "}
library(RTutor)
#library(restorepoint)
setwd("C:/folder_of_this_solution_file")
ps.name = "Example"; sol.file = paste0(ps.name,"_sol.Rmd")
libs = NULL # character vector of all packages you load in the problem set
#name.rmd.chunks(sol.file) # set auto chunk names in this file
create.ps(sol.file=sol.file, ps.name=ps.name, user.name=NULL,
          libs=libs, extra.code.file=NULL, var.txt.file=NULL)

# When you want to solve in the browser
show.ps(ps.name,launch.browser=TRUE, load.sav=FALSE,
        sample.solution=FALSE, is.solved=FALSE)

```
#>

## Exercise 1 -- Summary statistics

a) We often want to compute some summary statistic of a vector.
For example:
```{r "1 a)"}
#< task_notest
x = 10:20
# Computing the sum of x
sum(x)
#>
```

Now compute the mean of x.
```{r "1 a) 2"}
mean(x)
#< hint
display("Use Google, e.g. search for 'R compute mean'.")
#>
```

#< award "mean means mean"
Well, in some occasions one can just guess the name of an R function.
The function to compute the mean of a vector, or matrix is called 'mean'.
Usually, it is much quicker to goggle than to guess function names, however.
#>


## Exercise 2 -- Computing with vectors
```{r "2 "}
#< settings
import.var = c("x")
#>
```
````

```
a) Let y be a vector that contains the squared elements of x. Show y
```{r "2 a)"}
y = x^2
y
```

#< info "random numbers"
Here are examples for generating random numbers
```{r "1 "}
runif(3,0,100)
sample(1:100,5)
```
#>
```

The problem set folder of the RTutor library contains some example solution files.

1.4 Generate a structure file and empty problem set for students

Copy the solution file in some directory, and set is as working directory by changing in your solution file the line

```
setwd("C:/folder_of_this_solution_file")
```

Open the solution file in RStudio and press Ctrl-Alt-R to run the first code chunk.

The call to

```
create.ps(sol.file=sol.file, ps.name=ps.name, user.name=NULL,libs=libs)
```

generates the following 3 files - Example.rps: a binary representation of the problem set structure - Example.Rmd: empty problem set in Rmarkdown format - Example_sample_solution.Rmd: A sample solution of the problem set in Rmarkdown format.

Note: If you uncomment the line

```
#name.rmd.chunks(sol.file)
```

in the example above, the chunk names of your solution file will be automatically set to the naming convention in your empty problem set. This helps to navigate your solution file. To see the changes in your solution file in your RStudio editor window, you have to change the editor tab to some other file and then move back to the tab of your solution file. Unfortunately, sometimes RStudio will have problems to run all chunks with Ctrl-Alt-R if you have automatically renamed chunk names. If that happens try starting RStudio again.

1.5 Solving and checking the problem set

1.5.1 Solving and checking in the browser

The call

```
show.ps(ps.name,launch.browser=TRUE, load.sav=FALSE, sample.solution=FALSE, is.solved=FALSE)
```

should start your default browser with an empty problem set and you should be able to solve it by working on the first code chunk and pressing the "check" button when finished. If you set the argument `sample.solution=TRUE` the problem set is already filled with your sample solution. If you also set `is.solved=FALSE` all code chunks are already automatically checked when you start the problem set. Pressing the 'save' button on some code chunk in your problem set, will save your solution. You can load your previous solution by call show.shiny.ps with the argument `load.sav=TRUE`. This may not work, however, if you have modified the solution file. The browser version of the problem set is still some work in progress. It should work, but has bugs and needs generally some improvement. Suggestions and bug reports are welcome.

1.5.2 Solving and checking a Markdown file in RStudio

Alternatively a problem set can be solved by the students as an .Rmd file using RStudio. A student must copy the .rps file (the problem set structure generated by create.ps) and the empty problem set as R-markdown .Rmd file to a folder (same folder for both files) and open the .Rmd problem set file with RStudio.

The empty problem set files have some commands in the beginning, which are used to initialize RTutor and tell you, how you can check your solution. You first have to do 2 things:

1. Enter your user name at the specified place
2. Specify the folder in which you have copied this file

You can then start solving the problem set. To check your solution, you just have to

1. Save your .rmd problem set file (Ctrl-S)
2. and then run all chunks of your problem set (Ctrl-Alt-R)

How does it work? The header of the problem set file loads the package RTutor and calls the function check.problem.set, which runs automatic tests on the typed solution. If a test passes there is a success message. If a test fails, one gets an error message specifying in which part of the problem set the solution is not yet correct. Often one can type `hint()` in the R console to get a hint of how to solve the task that is not yet correct.

A student can type `stats()` to get some information of how much of the problem set he has already solved.

Before testing the empty problem set, open "Example_sample_solution.Rmd" and check if running all chunks (Ctrl-Alt-R) works correctly.

1.6 Iterative Development

I guess the most natural way to develop a problem set is to repeatedly perform the following 3 steps:

1. Write and adapt your solution file and generate problem set files with Ctrl-Alt-R
2. Test whether the sample solution runs without problems
3. Try to solve the empty problem set yourself and check whether you should change tests, hints or exercise tasks in your solution file (Step 1).

Advice: When Note that when you change your solution file and create new empty problem sets, all problem set files will be overwritten. When you solve the empty problem set during your iterative development, this can be annoying since you must replicate your solution steps again. Therefore it can sometimes be useful to save your solved problem set file, e.g. "Example.Rmd", under a different file name, like "Example_test.Rmd". You then must also adapt the line

```
ps.file = 'Example.Rmd' # this file
```

in the beginning of your problem set file to that new file name. Of course, once you generate new exercises in your solution file you want to use the newly generated problem set file.

1.7 Distributing your problem set

1.7.1 Simple way to distribute problem sets for a class

Distributing the problem set for RStudio use is simple: just give your students the structure file as .rps and the empty problem set as .rmd file and tell them to install RStudio and all packages as described above. Students can then go on solving the problem set. You can also ask them to upload the solutions and possible the log files that track their solution process.

It is planned to implement more functionality that facilitates grading. The goal is to put all solutions in a directory and quickly generate tables that show the total percentage solved for all students.

1.7.2 Distribute as an R package on Github

You can also create an R package that contains the problem set and host it on Github. This is described in more detail in this vignette:

https://github.com/skranz/RTutor/blob/master/vignettes/Deploy%20Problem%20Sets%20as%20Package%20on%20Github.Rmd

1.7.3 Host online as a shiny app on shinyapps.io

You can also host web-based problem sets on shinyapps.io. Users can then solve them without having to install R. This is explained in this vignette: https://github.com/skranz/RTutor/blob/master/vignettes/Deploy%20problem%20set%20on%20shinyapps.io.Rmd

In principle, you could also host the problem set as an app on your own shiny server. However, this seems a substantial security risk since users can run arbitrary R code, which allows them to access your file system and generate disaster. I still have to figure out in how far RAppArmor will allow safe usage on your own shiny servers. (I expect the RStudio team has figured out, how to make shinyapps.io save for them.)

2 Basic elements of solution files

Let us dig a bit deeper into writing solution files. Solution files are .Rmd files that basically specify the task with sample solution. Code chunks can contain special comments to customize tests or hints.

2.1 An example problem set

Let us start by just writing an exercise with solution, without yet adding some manual hints or tests. Here, we also neglect the code chunk before Exercise 1 that generates the problem set file.

````
# Problemset Example

#< ignore
```{r "Create problem set files"}
setwd("C:/folder_of_this_solution_file")
ps.name = "Example"; sol.file = paste0(ps.name,"_sol.Rmd")
libs = NULL # character vector of all packages you load in the problem set
create.ps(sol.file=sol.file, ps.name=ps.name, user.name=NULL,libs=libs)
```
#>
````

````
## Exercise 1 -- Summary statistics

a) We often want to compute some summary statistic of a vector.
For example:
```{r}
#< task
x = 10:20
# Computing the sum of x
sum(x)
#>
```
Now compute the mean of x.
```{r}
mean(x)
```
````

There are several keywords with special meaning. We have a problem set header starting with `# Problemset` an exercise header starting with `## Exercise`. There is an *ignore block* starting with the line `#< ignore` and ending with the line `#>` that contains a code chunk to generate the problem set files from this solution. Code or text within ignore blocks will not become part of the problem set itself.

The description of the exercise is written as standard text, the code of your solution is put in R code chunks. The first code chunk contains a *task block* starting with `#< task` and ending with `#>` that specifies that this code will be shown to the students as part of the task description of the problem set.

You can already generate an empty problem set from this simple solution file by running Ctrl-Alt-R (see step 3 of part I). In the empty problem set file, places where students have to enter code will be marked by.

````
```{r }
# enter your code here ...
```
````

We explain some syntax in detail more below.

2.2 Problemset header

Please always start your solution file with a line

```
# Problemset Example
```

where you replace `Example` with your own problem set name.

2.3 Exercise header

A solution file consists of one or more exercises. An exercise starts with a line like

```
## Exercise 1 -- Summary statistics
```

You can pick any label for the exercise (it can include spaces). Yet, the recommend format for the exercise name is an exercise number, optionally followed by `--` and a short description, as in the example above.

It is necessary that your line *must* starts with

```
## Exercise
```

2.4 parts a), b) ...

RTutor recognizes lines in the exercise description that start with a) or ii) etc. as a "part" of an exercise and will use this information when automatically naming the code chunks in the empty problem set.

2.5 task blocks

In our exercise the first code chunk shall be an example that will also be shown to the student. We make code being shown to students by wrapping it in a block starting with the line

```
#< task
```

and ending with a line

```
#>
```

Generate the problem set files and look at the empty problem set. You see that the code of the first chunk is now shown in the empty problem set while the second code chunk asks the student to enter her solution.

The interactive problem set can already be used. It automatically tests the solution and if the test fails since `mean(x)` is not entered correctly, it gives the option to type `hint()` to get a hint.

2.6 task_notest blocks

By default also the code as part of a task block will be tested for correctness when the student tests her solution. That is because future code may rely on the given code in the task and we want to make sure that the task code has not been accidentally changed by the user. If you don't want to generate tests for the code given in the task you can do so by wrapping the code instead in a block starting with `#< task_notest`, e.g.

```
{r eval=FALSE} #< task_notest sum(1:5) #>
```

2.7 Manual hints

RTutor tries to generate automatic hints that look at the students solution and try to give some advice while not telling too much. In particular, for very simple tasks the hint may reveal too much. For example, the automatic hint already tells the student the name of the function. (Often it is much harder to find the correct function arguments, though.)

```
hint()
```

```
You must call the function 'mean' in part a).
```

In our example, we want instead a manual hint that tells the student, how the internet is a very powerful source of information to find R commands for specific tasks. We include such a manual hint in our solution file as follows:

```
# Problemset Example

... chunk to generate problem sets is omitted ...

## Exercise 1 -- Summary statistics
```

````
a) We often want to compute some summary statistics of a vector.
For example:
```{r}
#< task
x = 10:20
# Computing the sum of x
sum(x)
#> task
```
Now compute the mean of x.
```{r}
mean(x)
#< hint
display("Use Google, e.g. search for 'R compute mean'.")
#>
```
````

This means we simply add after the command of the solution that will be tested, a block of the form

```
#< hint

#>
```

Inside the block, you enter some code that will be shown if the student types hint() after the test for the command failed. Often this will simply be a message, but you could also do other things, e.g. showing a plot as hint. Note that the hint code will be evaluated in an environment that contains all variables the student has defined earlier.

Sometimes you want to show the advice from the automatically generated hint but also want to add some additional advice. You can do this by putting code inside the following block:

```
#< add_to_hint

#>
```

2.8 Using variables from earlier exercises

By default variables that have been generated by the Student in earlier exercises are not known in the current exercise. Sometimes, we want to use variables from earlier exercises, however. The actual Example_sol.Rmd file contains a second exercise that uses the variable `x` from exercise 1:

````
## Exercise 2 -- Computing with vectors

```{r}
#< settings
import.var = list("1"="x")
#>
```
a) Let y be a vector that contains the squared elements of x. Show y
```{r}
y = x^2
y
```
````

As you see, in order to use variables from earlier exercises, you must add at the beginning of an exercise a settings block that defines a variable `import.var`, like

````
```{r}
#< settings
import.var = list("1"="x")
#>
```
````

The variable `import.var` is a list whose element names correspond to the short exercise names (part of the name before –) from which you want to import variables. The list elements are character vectors of the variable names you want to import from the corresponding exercise.

2.9 Info blocks

Info blocks are declared outside of code chunks and start with `#< info` and end with `#>`. They allow additional information that will be optionally shown as HTML in the RStudio viewer pane if the user wants to see it. They are very helpful to keep the main problem set text brief enough, while at the same time allow a wide range of detailed background information. Here is an example:

````
#< info "useful functions for numeric vectors"
  Here are examples for useful R functions
  ```{r "1 "}
  max(c(1,5,2)) # returns maximum
  min(c(1,5,2)) # returns minimum

  sum(c(1,5,2,NA), na.rm=TRUE) # returns sum of all numbers, ignore NA
  cumsum(c(1,5,2)) # returns cummulated sum
  diff(c(1,5,2)) # returns the vector of differences
  ```
#>
````

An info block can contain normal text and also code chunks. When the problem set is created all info blocks will be compiled via the knitr and markdown packages to html text. Since info blocks will be complied already when the problem set is created, you cannot used any variables that are declared outside the info block in the info block code chunks. If you want info blocks with code chunks that the user can solve, see the description of `notes` further below.

2.10 Giving awards

Isn't it amazing when video game players play for hours and hours, doing sometimes quite repetitive and boring tasks, just in order to earn some virtual points or virtual trophies? Well it seems that many people can be motivated, at least to some degree, by such stuff. Since a main motivation for RTutor is to make learning R, as well as statistical and economic concepts, more interesting and more entertaining, it seems natural to borrow some ideas from computer games.

So far there is only a small thing: students can get **awards** if they have solved a problem. The received awards can be shown by typing `awards()` in the R console. Information on the awards is stored in a file with name user_username.ruser in your problem set folder. You can give awards after having solved a chunk by adding after the chunk an award block (the block is declared outside of code chunks), e.g.

```
#< award "mean means mean"
Well, in some occasions one can just guess the name of an R function.
The function to compute the mean of a vector or matrix
is called 'mean'.
Usually, it is much quicker to goggle than
to guess function names, however.
#>
```

Just write some text inside the block. The title of the award must be set in "".

2.11 Adapting tests of student's solution

I tried my best to automatically test whether the student entered a correct solution or not. Yet, sometimes we need to manually include specific variants of tests in the solution file. Here is an example for such a manual test:

```
#' b) Save in the variable u a vector of 4 different numbers
u = c(3,6,7,99)
#< test
check.variable("u",c(3,6,7,99),values=FALSE)
#>
```

The automatic test would check whether one of the following two conditions is meet:

- u has the same value than in the solution. This means u=c(2,5,6,98)+1 would also pass as correct solution
- u is generated by an equivalent call as in the solution (equivalent means the function name should be the same and the arguments should have the same value). This is useful if the solution is a call that generates a random variable like x = runif(1).

Yet in this example, the automatic test is too restrictive. The student shall just generate some arbitrary vector consisting of 4 numbers. The block

```
#< test
check.variable("u",c(3,6,7,99),values=FALSE)
#>
```

replaces the automatic test with a test that just checks whether a variable u exists, and has the same length and class (numeric or integer) as an example solution c(3,6,7,99).

RTutor has a series of helper functions for such manual tests. Take a look at the package help for a documentation.

2.12 Specifying parameters of default tests

By default, tests are generated that call either `check.call` (a statement that does not assign a variable), `check.assign` (if a value is assigned to a variable), or `check.function` (if a function is generated). You can take a look at the generated _struc.r file to see which default and manual tests are generated. All test function have a number of arguments, that allow to customize the tests. A block starting with the line `#< test_arg` allows you to change the arguments of a default test. Consider the following example:

```
plot(x=p,y=q,main="A plot", xlab="Prices")
#< test_arg
  ignore.arg = c("main","xlab")
  allow.extra.arg=TRUE
#>
```

The `#< test_arg` block customizes the parameters `ignore.arg` and `allow.extra.arg` of the `check.call` function. The parameter `ignore.arg = c("main","xlab")` means that the student does not have to add these two arguments to the plot function or can use different values. The parameter `allow.extra.arg=TRUE` allows the student to specify additional arguments when calling plot, e.g. specifying a `ylab`. So essentially, it will now only be tested whether the x and y arguments of the plot are correct and any customization of the plot will still be considered a correct solution. See the help of `check.call` for a description of arguments.

If the sample solution assigns a value to a variable, the default function is `check.assign`. See its help file for possible arguments.

3 Examples and tipps for creating tests and hints

3.1 Testing a function written by the student

Sometimes you may ask to write a function in the problem set. Here is an example, how you could construct a solution file. In the exercise the student is asked to complete a manual function `ols` that shall compute an ols estimate:

```
#< task_notest
ols = function(y,X) {

  # enter code to compute beta.hat here ...

  return(as.numeric(beta.hat))
}
#> task

ols <- function(y,X) {
  beta.hat = solve(t(X) %*% X) %*% t(X) %*% y
  return(as.numeric(beta.hat))
}
#< test_arg
  ols(c(100,50,20,60),cbind(1,c(20,30,15,20)))
  hint.name="ols"
#>
#< hint
display("Just insert inside the function ols the code to compute beta.hat",
        "from y and X. You have developed this code in Exercise 1.")
#>
```

First, we have a `#< task_notest` block that specifies an unfinished function that will be given to the student. Afterward, we have an example of a correct function `ols`. Then the `#< test_arg` block specifies parameters for the automatic test `check.function`. The unnamed parameter

```
ols(c(100,50,20,60),cbind(1,c(20,30,15,20)))
```

Specifies a test call. `check.function` will run this test call for the official solution and the student's solution. The test will only pass if both versions return the same value. The parameter hint.name specifies a manual name for a hint, since currently no automatic hints are generated for function creation (hopefully that will improve in later versions of RTutor.) Finally, the `#< hint ols` specifies what the hint `ols` shows.

3.2 Testing a function that generates random variables

In the moment there is relatively little support to check user written functions that generate random variables, and I hope this will improve with newer versions of RTutor. What currently can be done to compare the results of the student's function and the official solution using the same random seed.

Here is an example:

````
a) Write a function `runif.square` with parameters n, min and max
that generates n random variables that are the square of variables
that are uniform distributed on the interval from min to max.

```{r}
runif.square = function(n,min,max) {
  runif(n,min,max)^2
}
#< test_arg
  with.random.seed(runif.square(n=20,min=4,max=9), seed=12345)
#>
```
````

Our test call is now embedded in the function with.random.seed that calls the function with a fixed random seed. The test check.function only passes if the students function and official solution return the same value when called with the same random seed.

If a function requires simulation of more than one random number, this testing procedure only works if the student draws the random numbers in the same order than the official solution. This means your task should specify already a lot of structure for the function and tell the student not to draw any additional random variables inside the function.

3.3 Test if variables that satisfy specific conditions are generated

Consider the following example:

````
a) Specify a number of observations T>=5

```{r }
T = 100
#< test
  check.variable("T",100,values=FALSE)
  holds.true(T>=5, failure.message="You must set T>=5.")
#>
```
````

The default test `check.assign` would only pass if the student would set `T=100`, but any T>=5 shall be considered correct. So we first replace the default test by a `check.variable` test that checks whether a numeric variable T of length 1 exists. Then we add another manual test `holds.true` that checks whether T>=5 holds. Note that using the holds.true test without first checking that a numeric, variable T of length 1 exists can cause the holds.true test to crash.

4 Some more advanced features

4.1 dplyr chains with pipe operator %>%

The pipe operator `%>%` allows to create command chains. This is particularly often used with dplyr verbs. For example:

````
```{r "5 d)"}
d %>%
  group_by(ye,ma) %>%
  summarise(Q=sum(qu)/1e6) %>%
  arrange(-ye,Q)
```
````

RTutor tries to automatically generate useful hints, if the user makes an error in such chains. The hints try to establish where in the chain the error has occured and to guide the user step by step. This does not (yet) work perfectly in all instances, however. (For example, errors in group by are typically detected only later, so far). While you cannot specify manual hints for particular steps in the chain, you can add a global hint with an `<# add_to_hint` block.

4.2 Check computations that involve several steps with compute blocks

Sometimes you want to ask students to perform computations that will usually require several steps. One example are dplyr chains, as explained above. But there are other forms of multistep computations. Two somewhat opposite ways of implementing such multistep computations in a problem set would be the following:

i) Only check the final result and let the student figure out the intermediate steps herself.
ii) Include all intermediate steps as part of the problem set

A `#< compute ... #>` block allows an intermediate approach. Here is an example from a problem set of mine that asks a student to compute a matrix of choice probabilities from a conditional logit model.

```
#< compute P

## Take a look at the formula for the choice probabilities P.nj of the logit discrete choice model

## Let exp.V be a matrix that contains the numerators of P.nj (use scale as sigma in your formula)
exp.V = exp(V/scale)

## Let sum.exp.V be a vector that contains for each person n the denominator in P.nj. You can use t
sum.exp.V = rowSums(exp.V)

## Compute P as ratio of the numerator and the denominator
P = exp.V / sum.exp.V
#>
```

The solution will pass as correct if the final values of `P` are correct. The student is not obliged to perform the particular intermediate computations like `exp.V`. Yet, if the student has not yet correctly computed `P` and types `hint()`, the hint function tries to steer the student step by step through the sample solution described in the block. The comments starting with `##` will be transformed into text that will be shown in the hint.

4.3 Chunks with HTML output

You can specify chunks whose output is native HTML code, e.g. if you want to show an HTML table or show a motion plot. As in normal rmarkdown, you can do so by setting the chunk option `results= "asis"`. Here is an example from an interactive problem set that shows a motion plot:

````
```{r "2_1_a_example", results = 'asis'}
#< task
library(googleVis)
mp = gvisMotionChart(Int.data,
       idvar = "country", timevar = "year")
plot(mp, tag = "chart")
#>
```
````

There are a lot of R packages that help creating nice HTML tables, like `stargazer`, or `hwriter`.

4.4 Specifying figure width and height

If your chunk plots a figure, you may want to specify the width and height of the resulting figure. You can do this with the chunk options `fig.width` and `fig.height` that specify width and height in inches.

````
```{r "1_a", fig.width=12, fig.height=5}
#< task
plot(1:10,(1:10)^2)
#>
```
````

You can also specify chunk options of the `knitr` package. An overview is given here:

http://yihui.name/knitr/options/

4.5 Optional chunks

By default you have to solve all chunks in an exercise in the given order. If you set the chunk option `optional=TRUE`, a chunk may not be solved. Example:

````
```{r "1_a", optional = TRUE}
# Code here

```
````

This is in particular useful if you use notes that contain chunks. Note that variables that have been computed in optional chunks will not be known in later chunks.

Note that `optional` is not a chunk option known by `knitr` but only used for RTutor.

4.6 Notes – Or info blocks that can contain chunks

The simplest way to add notes / footnotes is to use an info block as described above, with a block:

```
#< info "Title"
Description here and code that will be compiled beforehand.
#>
```

Info blocks cannot include chunks, however. There is a new feature of RTutor that allows notes that contain optional chunks. Here is an example:

````
#! start_note "A note with a chunk"
This is a note.

```{r "1_1  6",optional=TRUE}
#< task
# show that all integers between 0 and 10
#>
1:10
```
#! end_note
````

Everything between the lines

`#! start_note "Title"`

and

`#! end_note`

will be shown in a note. Chunks inside the note can be solved by users. They should have `optional=TRUE` in the chunk header. Beware that RTutor is not (yet) very good in giving sensible failure messages, if you make mistakes, like forgetting the `#! end_note` line.