Data.frames, indexing and sub-setting

20160615:12.54

On completion of this module you will:

- Be able to write the contents of a data.frame out to a text file
- Be able to import a local text file into R to create a data.frame
- Be familiar with the functions names() and str() and will know how and when to use them
- Know how to rename the columns in a data.frame
- Be able to demonstrate an understanding of the row / column table structure of data frames and the essential differences between rows and columns
- Understand and be able to use $ notation for data.frame column names
- Understand and be able to use simple indexing as it applies to vectors and also single data.frame columns
- Understand and be able to use simple row / column indexing on data.frames
- Understand and be able to build more detailed row expressions to select subsets of population with particular characteristics
- Know how to add columns to a data.frame
- Know how to remove columns from a data.frame
- Know how to re-arrange the order of columns in a data-frame

Preamble

- It is suggested that you use <ALT> + <TAB> to alternate between this Workbook module and RStudio
- In this module where a series of commands open drop down menus or dialogue boxes we will provide the sequence to follow separated by the pipe "|" character (e.g. **Files |New File | R Script** to open up a new R script)
- This module assumes that you have already studied the modules **"Introducing RStudio"** and **"Creating vectors and working with them"** or that you have a commensurate level of understanding of R and RStudio. You can review the learning content of these modules from the Workbook index pages. If you are in any doubt we would recommend that you study these modules before proceeding any further
- You will also need to reload the environment file **WBenv1.RData** that was created and then saved in the module **"Introducing RStudio".** It

should be in a folder called **RStud-master**. If this folder does not exist then you should first clean out the Environment (see below) and then follow instructions to be found here to recreate that environment and to populate the **RStud-master** folder

Initial setup

Assuming that you already have RStud-master folder in place please carry out the following:

- Clean the Environment (Click here for help)
- Click on **Files** tab and navigate to the **RStud-master** folder
- Open this and set it as your Working Directory (hint – click on **More**)
- Click on the file **WBenv1.RData** and then on **"Yes"** to confirm

Writing out the contents of a data.frame to a text file

We will start this module by writing out the contents of a data.frame to a text file. Later we will then use the resulting CSV file to show how data rom a local text file can be very simply imported into a data.frame. To create the CSV file:

Create a new script and type into it (or copy and paste) the following four lines:

```
###Dataframe script
##
###Write WBdata out to CSV file
write.csv(WBdata,"WBdata.csv",row.names=FALSE)
rm(WBdata)
```

Run these four lines to write the contents of the WBdata data.frame to a CSV file in your working directory and also to delete the data.frame WBdata (see here for more detailed explanation). We will using the CSV file to re-create the data.frame.

Note, in passing, the use of the function **rm** which deletes objects from the environment. It is possible to delete multiple objects in one go by simply separating the object names with a comma

Save the script named as 'Dataframe script'

Importing data into a data.frame

The simplest way to import data into a data.frame is through the use of the RStudio 'Import dataset' facility. The tab for this can be found under the Environment panel tab. Clicking on this tab reveals the two options "From Local File" and "From Web URL. We used the Web URL option in the module "Introduction to RStudio". This time we will import a locally held csv file.

Click on the 'Import dataset' tab
Select "From Local File" option.
In the Select File to Import dialogue box either click on WBdata.csv or type in the filename
Click 'Open'.

The Import Dataset wizard opens.
On the right
The data from the CSV file is displayed in the upper box. In the lower box we can see how that data will look in the data frame.

On the left
The **Name** box automatically picks up the filename but if necessary this name can be changed
Encoding is best left as automatic
Heading determines whether or not the first row should be processed as column names. In this case note that 'yes' should be checked
Row names can usually be left as automatic
Separator. This is a CSV file so that Comma is the appropriate field separator. There will be occasions when a different character will be needed
Decimal point can usually be left to 'period'
Quote is not always present but in this case 'double quote' is appropriate
Comment is normally left to its default 'none'
Na strings. This refers to representation of absent / null values. Leave as'NA'
Strings as factors. We will consider factors later. In the meantime ensure that this box is checked

Typically all or most of the default settings of the import wizard can be accepted but try changing some of the above settings to see the effect on the data in the lower box. When finished, click on the 'Import' tab.

The **WBdata** data.frame is restored and will have opened in the Source panel. This can be closed.

Any text file can be imported into R in this way simply by navigating to the file location and then tailoring the import wizard settings to suit the file.

Inspecting the data.frame WBdata using the functions names and str

In the 'Dataframe script' type or copy and paste the following line:

```
names(WBdata)
```

Run this line and notice that R returns its evaluation in the console

names function

The **names ()** function is very useful. It can be used either to **get** or to **set** the names of an R object. In this case it gets the column names of the dataframe. At present these column names are not of much use. We can now use the same **names ()** function to set the column names to more useful and meaningful values.

We will do this first by setting up a vector variable that holds the column names that we want, in the right order. We will then assign the values in the vector to the data.frame column names using the **names()** function

First step: Add a line to your script to assign the following values

```
"Id      ","Age","GenderMF","Hypert","SBP","DBP"
```

to a vector variable called colnames. (Hint – this can be done using **c()** to assign these six values to a vector object called 'colnames').
Do this now.

Second step: Assign the names in this vector to the dataframe column names. To do this, insert the following two lines into your script and run them:

```
names(WBdata) <- colnames
names(WBdata)
```

This time the names function returns the new column names.

We have now easily and quickly restored the WBdata data.frame by importing a csv file and have then and given it meaningful column names.

str function

The **str** function 'compactly' displays the structure of an R object. Insert the following line into your script and run it:

str(WBdata)

In the console panel R will respond with something similar to this:

```
$ Id       : int  10001 10002 10003 10004 10005 10006 10007 10008 10009...
$ Age      : int  74 22 11 69 79 24 6 5 5 22 ...
$ GenderMF: Factor w/ 2 levels "F","M": 1 1 2 2 1 2 1 2 2 2 ...
$ Hypert   : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
$ SBP      : int  134 111 NA 111 150 NA NA NA NA NA ...
$ DBP      : int  87 75 NA 70 62 NA NA NA NA NA ...
```

The top line indicates that this is a data.frame with 10000 observations and six variables. Each subsequent line (starting with a "$" sign) provides information about each of six columns, giving the name, data type, and then a list of the first few components.

The **str** function can therefore be very useful in giving an overview of the structure of a data.frame and also of other R objects.

Data.frames: row / column structure

A data.frame is used for storing data tables. It is essentially a two dimensional structure arranged in rows and columns. Reading down a column individual cells will all be of the same data type. In contrast, reading along a row individual cells may be, and often are, of different data types. A data.frame can be defined as a list of vectors all of the same length.

Clinical data used in research is typically tabular and has a "one row per patient" characteristic. All of the characteristics in a given row relate to one patient and there will typically be a pseudonym unique for each patient at the start of each row. Each row 'cuts' across all of the columns in a table and each column will typically represent one characteristic. Individual columns will each have different data types depending on the characteristics that they represent. Some tables will have large numbers of different columns and there may be thousands or even millions of rows. Much of the action around data analysis involves manipulating this basic row / column structure by means of selective indexing

Data.frames: Viewing the data

This section briefly outlines some very simple ways to view the data held in a data.frame.

The quickest way to view the data held in the data.frame is to open it in the Source panel by clicking on the data.frame object in the top right Environment panel. Depending on the size of the data.frame not all of the rows will necessarily be displayed.

The **View()** function (note upper case 'V') either in a script or typed into the Console will also open a data.frame in the Source panel. For example, try adding to your script and running the line:

```
View(WBdata)
```

The **head()** function will display the first 6 rows of a data.frame with column names, in the Console panel:

```
head(WBdata)
```

The **tail()** function will display the last 6 rows of a data.frame with column names, in the Console panel :

```
tail(WBdata)
```

Data.frames: $ notation for individual columns

As we saw above, the **str()** function lists the columns in a data.frame with names prefixed by the "$" sign. This **"$" notation** is one of a number of useful ways by which the values in a given column can be pulled out of a data.frame. Type in to your script and run the following line. Use the <TAB> key to complete:

```
WBdata$Age
```

Note how when you suffix WBdata with the "$" sign RStudio helpfully gives you a complete list of all of the column names in the WBdata data.frame. You can click on the column name that you want to auto-complete

When you run the line, R responds by listing out all of the ages in the Age column. Note the figure in square brackets at the start of each line. This number is the **index number** of the cell immediately to its right. Index numbers in R always appear in square brackets. Every cell in a column has a unique index number just as it is true that every component of a vector has a unique index number. We will look at indexing in more detail later.

What is the length of the column WBdata$Age?

This can be found by using the **length()** function. Type into your script and run the following line:

```
length(WBdata$Age)
```

Repeat the above exercises for other column names in WBdata to familiarise yourself with $ notation, the RStudio auto-completion feature for column names, and also to confirm that all of the columns in this data.frame do indeed have the same length.

Data.frames: simple column indexing

We will start by looking at indexing as it applies to the simple **colnames** vector that we created earlier. Type into your script and run the following line:

```
str(colnames)
```

In the console panel you will see something similar to:

```
> str(colnames)
 chr [1:6] "Id" "Age" "GenderMF" "Hypert" "SBP" "DBP"
```

The **str()** function tells us that this is a variable of type chr. It is in fact a vector. The square bracketed[1:6] indicates that colnames has components with index numbers from 1 to 6. The key point here is that we can use the indexing to get just the components that we want. Type into your script and run the following line:

```
colnames[3]
```

Note that R returns the third component in colnames. We can enter any combination of indexing. Try for example

```
colnames[2:4]
```

or

```
colnames[c(1,3,5)]
```

Remember that the columns in a data.frame are effectively vectors so this means of selective indexing will also work on **WBdata$Age**.

Try experimenting with different combinations of indexing in square brackets after **WBdata$Age** or any other of the data.frame columns.

So far we have looked at indexing in vectors and single data.frame columns. How do we incorporate the rows into indexing?

Data.frames: row / column indexing

Row / column indexing clearly requires two sets of values, one to define the rows that we want and the other to define the columns that we want. In a table we can pin point any single cell simply by providing its row number and its column number; this essentially works on the same principles as for any map reference. This uses indexing and so again involves the use of square brackets and a comma **[,]**. There is a very simple convention; expressions to the left of the comma define rows and expressions to the right of the column define columns. There is a further extension to this convention. Where the row expression is left blank R assumes that ALL rows will be selected. Where the column expression is left blank R assumes that ALL columns will be selected.

To find the age of the patient represented in the second row we need to know the column number (index) for Age. We can get this by using **names(WBdata)** where we find that Age has index 2 Then we can do:

```
WBdata[2,2]
```

As with the notation used with single columns / vectors the expressions for rows and columns can be very detailed. Here are some examples to try. Copy and paste these lines and run them:

```
WBdata[1:100,]          #rows 1 to 100 and all columns
WBdata[,1:3]            #all rows with columns 1 to 3

WBdata[c(1:10,15,20),1:3]       #rows 1 to 10, 15 and 20 with columns 1 to 3
```

Try experimenting with different combinations of rows and columns

Data.frames: using row column indexing expressions to select subgroup of interest

So far we have simply used row and column numbers but typically we will want to select groups of patients with certain characteristics – for example to find cases or particular outcomes. Suppose that we want to find the population that has hypertension and to create a new data.frame called Hypertdf to hold this population. To do this we will want to find all rows where it is TRUE that Hypert is 'Yes'. Remember that R evaluates any expression that is offered to it and in many cases this involves determining whether an expression holds as TRUE or FALSE

We can build the expression that we need in steps. Copy these lines into your script and run them one by one:

```
WBdata$Hypert=="Yes"                          #Line 1
WBdata[WBdata$Hypert=="Yes",]                 #Line 2
Hypertdf <- WBdata[WBdata$Hypert=="Yes",]  #Line 3
```

Explanation:

- Line 1
  (Note the "==" logical operator used here for testing whether or not it is TRUE for each individual value in **WBdata$Hypert** that it equates with "Yes")
  R evaluates this expression for every row and returns TRUE or FALSE for each row instance of **WBdata$Hypert**
- Line 2
  To the left of the comma we have the expression from line 1. To the right of the comma we have a blank
  R evaluates the row expression and ONLY returns those rows where the row expression is TRUE
  R finds no column expression and so returns ALL columns
- Line 3
  Is a repeat of Line 2 but this time the output is assigned to a new variable called Hypertdf
  R assumes that this variable will be a data.frame because it is being fed with multiple variables of same length. It automatically adopts the column names from the original data.frame

Note: we recommend that you spend some time on building your confidence with this use of indexing and experiment with your own selections.

Further suggested exercises:

1. Limit the population to males only
2. Limit the population to age over 60
3. Limit the population to those with hypertension, who are male, and over 60 years – click here for help

Data.frames: Adding new columns

This can be done very simply using the $ convention. Suppose that we want to add a new column called Diagnosis to our Hypert data.frame and that we wanted to set all the values to "Hypert". Add the next two lines to your script and run them:

```
Hypertdf$Diagnosis <- "Hypert"
str(Hypertdf)
```

The **str()** function shows us that the first line has added a new column called Diagnosis with data type chr and has assigned the value "Hypert" to each of its components

It is possible to build an expression that does a calculation for example using the values in existing columns. Using the same $ convention we could create another column called BPPulsePressure and use this to hold the calculated difference between systolic and diastolic BPs for each row: Copy the following two lines to your script and run them:

```
Hypertdf$BPPulsePressure  <- Hypertdf$SBP - Hypertdf$DBP
str(Hypertdf)
```

This time we have a new column with datatype Int and with a varying set of values for each row.

(Just as a matter of interest we can quickly digress to run the **summary()** function on this new column to check its minimum and maximum values. We can also look to see whether there is any evidence for a change in the BP pulse pressure with age in this group by using **plot()** to generate a density plot:

```
summary(Hypertdf$BPPulsePressure)
plot(Hypertdf$Age,Hypertdf$BPPulsePressure)
```

It is at least reassuring that there are no negative values – i.e. cases where SBP and DBP have become reversed. The density plot does not suggest any great correlation with age)

Data.frames: Removing columns from a data.frame

There are at least two simple ways to remove columns from a data.frame but first, we will produce a replica of the **Hypertdf** data.frame so that we can experiment on this without affecting the original data.frame. We can call this replica data.frame **Hypert2df**. This can be very easily done by running the following two script lines:

```
Hypert2df <- Hypertdf
str(Hypert2df)
```

The **str()** function confirms that Hypert2df has the same structure as our original data.frame

Using $ naming convention

The simplest method for removing a column from a data.frame uses the $ naming convention. A column can be removed simply by assigning the value NULL to it. For example, to remove the column named Hypert we can run the following two script lines:

```
Hypert2df$Hypert <- NULL
str(Hypert2df)
```

The **str()** function confirms that **Hypert2df$Hypert** column has been removed.

Exercise: Try removing further columns from the Hypert2df data.frame. Note that you can regenerate the data.frame at any time by re-running the appropriate script lines above.

Using row column indexing

A more elegant way of removing columns is to use indexing and to assign selected columns to a new data.frame. Not only is it possible to decide which columns should and should not be included in a new data.frame we can also re-arrange the column order. Suppose that once again we want to remove the

column called “Hypert” but this time, in addition, we want to move the new “Diagnosis” column into its place.

We can start by running the names() function on the original Hypertdf data.frame to get the relevant column index numbers:

```
names(Hypertdf)
```

There are 8 columns in this data.frame. We want to exclude the column with index 4 and move the column with index 7 into its place. We therefore want to build a new data.frame with columns with the following index numbers 1, 2, 3, 7, 5, 6, 8 and in that precise order. We want to preserve all of the rows.

Using row column indexing we therefore want to leave the row expression blank and build a column expression including the column indices that we want – using **c()**. The column expression that we need will therefore be: **c(1, 2, 3, 7, 5, 6, 8)**. Putting this all together into one script line and assigning the results to **Hypert2df** data.frame we can do:

```
Hypert2df <- Hypertdf[,c(1,2,3,7,5,6,8)]
str(Hypert2df)
```

The **str()** function confirms that we have indeed removed the old “Hypert” column and replaced it with our new “Diagnosis” column – effectively re-ordering the original **Hypertdf** data.frame columns

Exercise: Try different combinations of column index numbers

Footnote about sub setting

The Module Dataframes, Indexing and Subsetting ends here.

Instructions for setting up the environment

First download the data file WBdata.csv

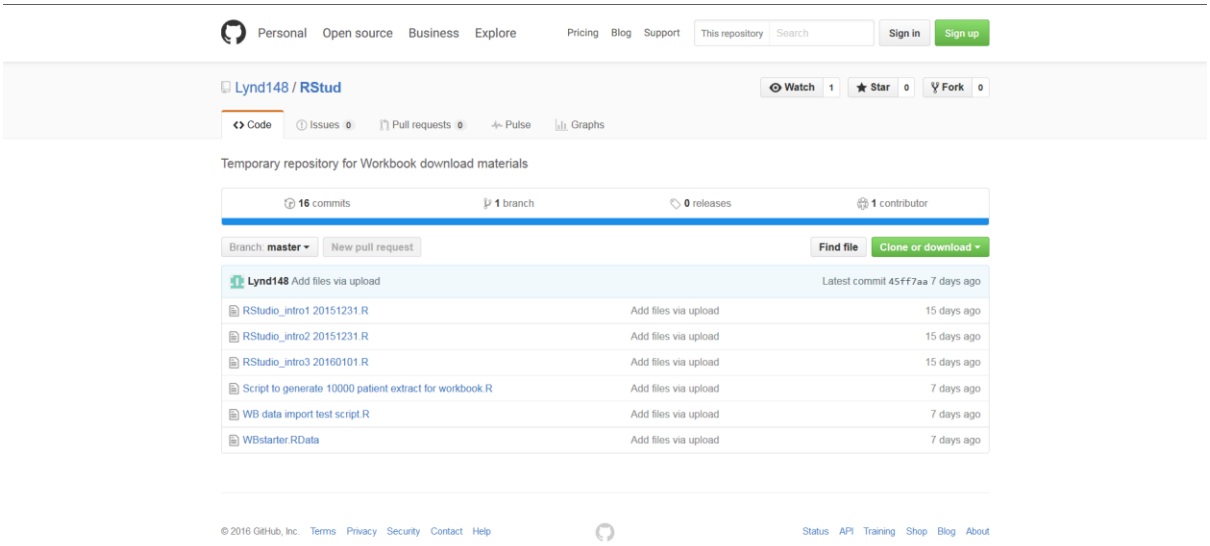
1. Go to the Environment panel (top right with Environment tab active)
2. First, clean out the environment by clicking on the broom icon and then clicking “Yes” in the “Confirm Remove Objects” dialogue box
3. Then, just below and to the Right of the Files tab find the “Import Dataset” tab
4. Click on the Import Dataset tab and select “From Web URL”
5. This will open the “Import from Web URL” box
6. Either copy and paste or type in the following URL
   https://raw.githubusercontent.com/Lynd148/WBdata/master/WBdata.csv
7. Click on ‘OK’
8. The Import dataset wizard box will open. Accept all of the default settings
9. Click on Import to create the data.frame WBdata
10. Close the WBdata view in the Source panel

Next download the scripts and Rdata resources from the Course web repository. Select the appropriate version below
Version for Desktop / notebook based RStudio
Version for Server based RStudio

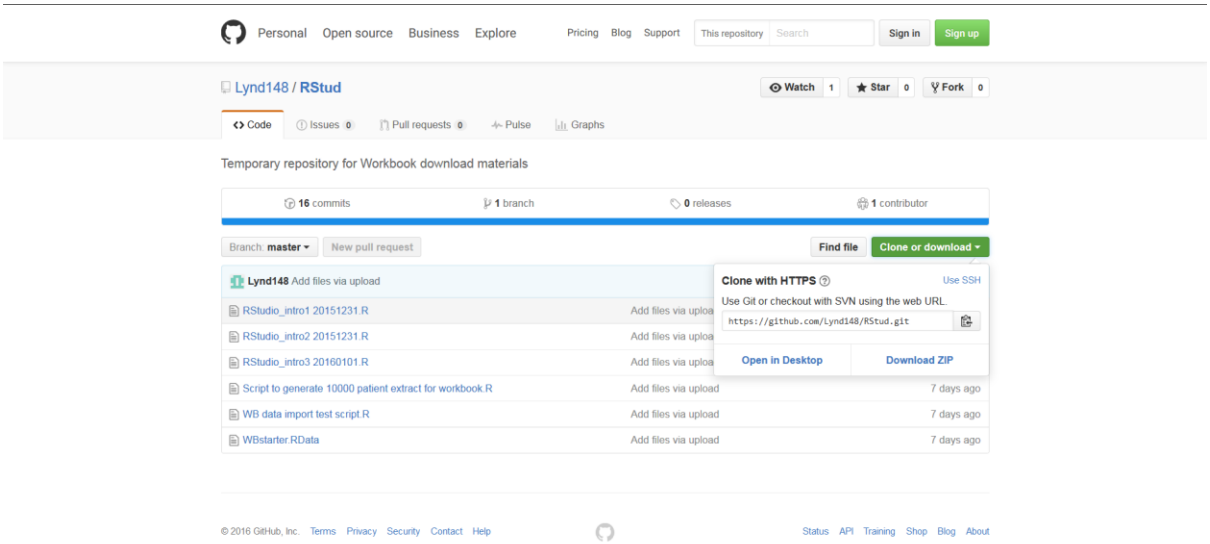
File download for Desktop / notebook based RStudio

1. Open you browser and either copy and paste the following URL to the address bar or type it in: **https://github.com/Lynd148/RStud.git**

2. This should open up a github repository with the title “Temporary repository for Workbook download materials” which will look something like this and contain a number of files. Note the button marked “Clone or Download” below the blue line

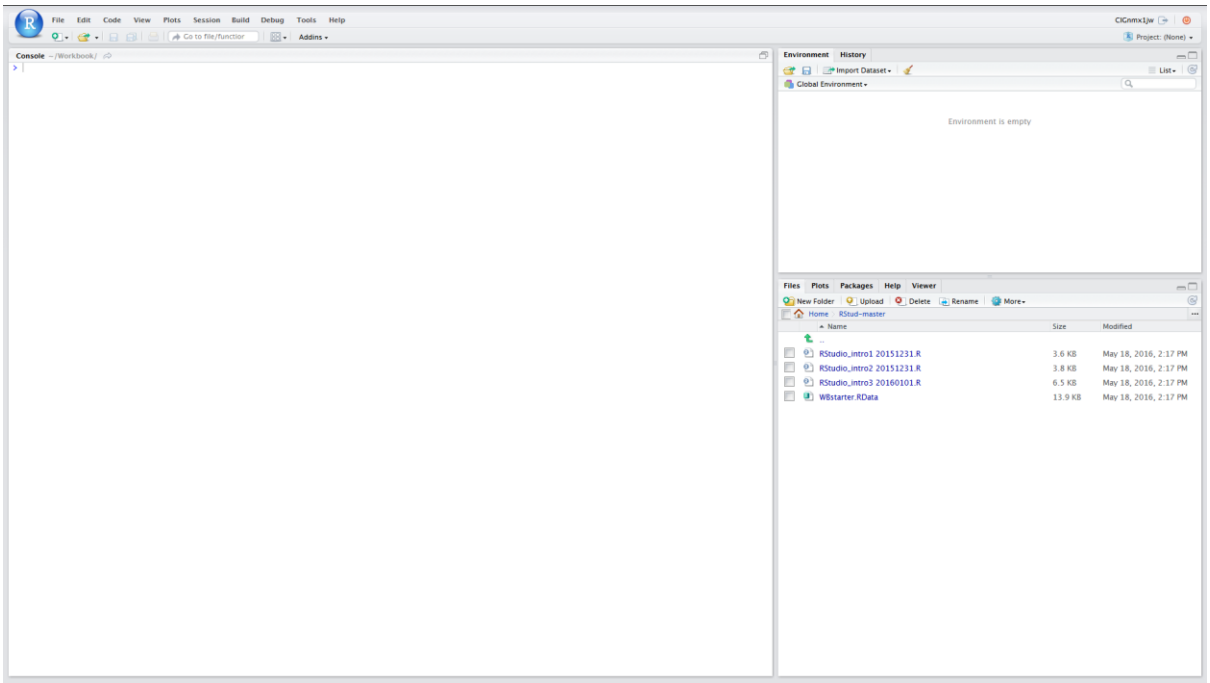


3. Click on the “Clone or Download” button. A dialogue box will open with two options: “Open in Desktop” and “Download ZIP



4. Click on the Download ZIP button. Depending on your browser and settings this will either open a dialogue box asking whether you want to open or save a file called RStud-master.zip or else will result in this file being immediately downloaded to your download folder. In the former case select 'open' and if necessary change the "Open with" option to Windows Explorer then click "OK". The folder RStud-master should appear after being downloaded

   In the latter case (where the file has automatically downloaded) navigate to your download folder. Right click on the file and select 'open with' and the option Windows Explorer. The folder RStud-master should now appear
5. Copy the folder RStud-master to your C-drive or other easily found destination
6. Now return to RStudio. Make sure that the Files tab of the bottom right Files Plots Packages Help Viewer panel is active and click on the small box at the right edge of the screen containing three dots …
7. This should open a "Browse for folder" box. Browse to the folder RStud-master on your C-drive or wherever else you may have placed it and click OK.
8. The folder will now open under the Files tab which should look similar to the screen shot below



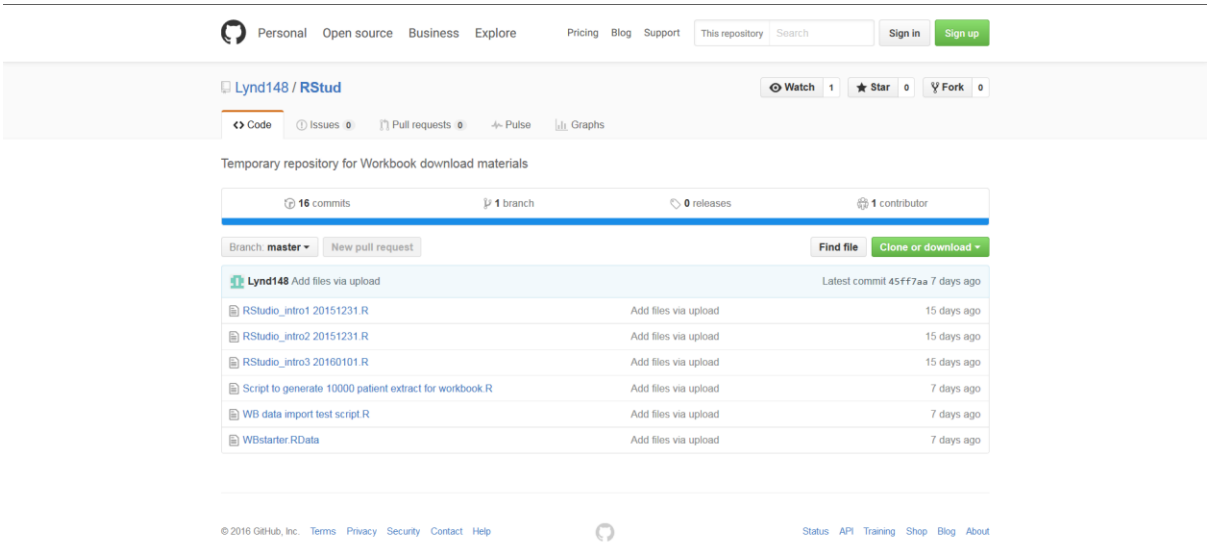
9. We now just need to set up the Working Directory. To do this click on the "More" tab (next to blue cogwheel icon just below the Files Plots Packages Help Viewer bar) and then Click on "Set as Working Directory"
10. All of the scripts and other files that we need in order to proceed should now be downloaded and also you have set the working directory
11. Click on the file WBstarter.RData to load it into the environment
12. Now save the environment to **WBenv1** click on the Environment tab and then the blue floppy disk item, enter **WBenv1** into the dialogue box and click Save

Click here to return to start of this module

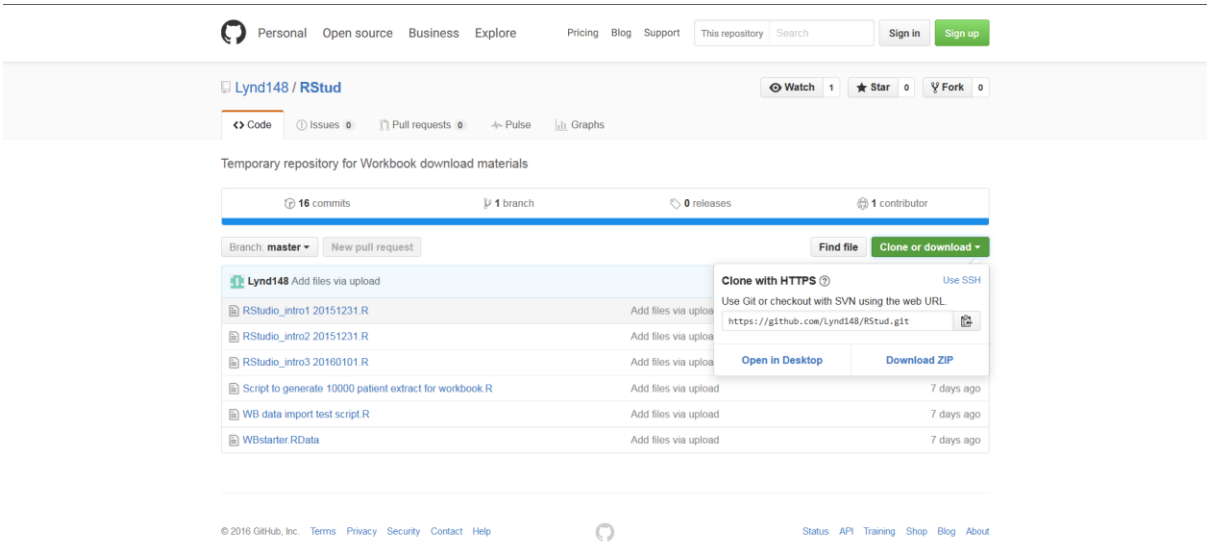
File download for server based RStudio

1. Open you browser and either copy and paste the following URL to the address bar or type it in: **https://github.com/Lynd148/RStud.git**

2. This should open up a github repository with the title "Temporary repository for Workbook download materials" which will look something like this and contain a number of files. Note the button marked "Clone or Download" below the blue line



3. Click on the “Clone or Download” button. A dialogue box will open with two options: “Open in Desktop” and “Download ZIP”



4. Click on the Download ZIP option. Depending on your browser and settings this will either open a dialogue box asking whether you want to open or save a file called RStud-master.zip or else will result in this file

being immediately downloaded to your downloads folder. In the former case select 'save' and click "OK" because in this scenario we want a ZIP file to be placed in your download folder

5. Having ascertained the location of your downloads folder return to RStudio. Make sure that the Files tab of the bottom right Files Plots Packages Help Viewer panel is active.
6. Next you need to set your home directory to be the Working Directory
    a. Click on Home
    b. Click on More (next to blue cogwheel icon) and then Click on "Set as Working Directory"
7. Now look for the Upload button just below the Files Plots Packages Help Viewer bar. Click on this
8. This should open a "Upload Files" box. Within that box browse to the destination folder for your downloads and find the file RStud-master.zip and select / open this and then click on OK
9. This should result in the folder RStud-master being added to your home directory
10. Open this
11. Next you need to reset your home directory to this RStud-master directory. As before, click on the "More" tab (next to blue cogwheel icon) and then Click on "Set as Working Directory"
12. All of the scripts and other files that we need in order to proceed should now be downloaded and also you have set the working directory.
13. Click on the file WBstarter.RData to load it into the environment
14. Now save the environment to **WBenv1** click on the Environment tab and then the blue floppy disk item, enter **WBenv1** into the dialogue box and click Save

Click here to return to start of this module

Lookup

Set Working directory
File | More | Set as Working Directory
Click **<ALT> + Left Arrow** <- to return

To clear the environment:
Click on the Environment tab (upper left panel)
Click on the broom icon
In the Confirm Remove Objects dialogue box click 'Yes'
Click **<ALT> + Left Arrow** <- to return

To set up colnames vector:
colnames <- c("Id","Age","GenderMF","Hypert","SBP","DBP")
Click **<ALT> + Left Arrow** <- to return

Write out a data.frame to CSV file
If you use 'help' on the function **write.csv** you will see that it takes a number of arguments. With the exception of the first two arguments the remainder are set to default values. Therefore as a minimum, this function needs:

- As its first argument the name of the data.frame to be written out
- As its second argument the name of the file to which the contents should be written out
- Data.frames have row names as well as Column names and the default is for both of these to be written out. The assumption is that the first row will provide the column names – which in this case we want. We do not want the row names which at the time that a data.frame is created typically default to a rendition of the row numbers. Hence the argument row.names=FALSE

Click **ALT> + Left Arrow** <- to return

Help with row / column indexing

To limit the population to males only, use:

```
WBdata$GenderMF=="M"
```

To limit the population to age over 60, use:

```
WBdata$Age>60
```

To limit the population to those with hypertension, who are male, and over 60 years age, use "&" operator to combine expressions:

```
Hypertdf <- WBdata[WBdata$Hypert=="Yes" &
WBdata$GenderMF=="M" &
WBdata$Age>60,]   #line 3
```

For more detail about logical operators see **Introducing RStudio** module

Click **ALT> + Left Arrow** <- to return

Subsetting using the R function **subset()**

A web search on 'R subset' will reveal this function and instructions about its use. It can make the creation of subsets much easier than by using [,] row column indexing.

This function has been deliberately omitted from this module for two reasons. Firstly, one the objectives of this module is to explain row / column indexing. Secondly, expert R users sound caution in its use and indicate that its arguments (for example 'select') can be confusing and that in certain settings it produces unexpected results. See for example: http://stackoverflow.com/questions/9860090/in-r-why-is-better-than-subset

We would therefore advocate using row / column indexing rather than **subset()**. If you do use the **subset()** function – then do so with caution

Click **ALT> + Left Arrow** <- to return