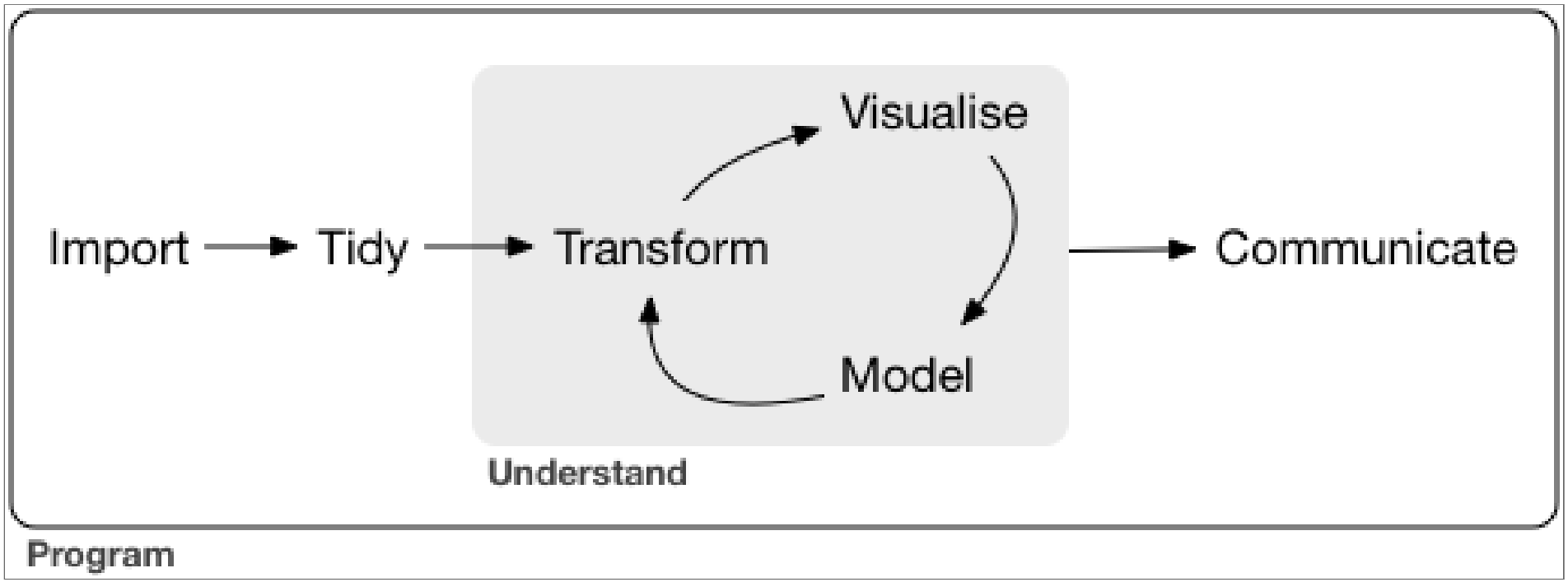
Introduction to R

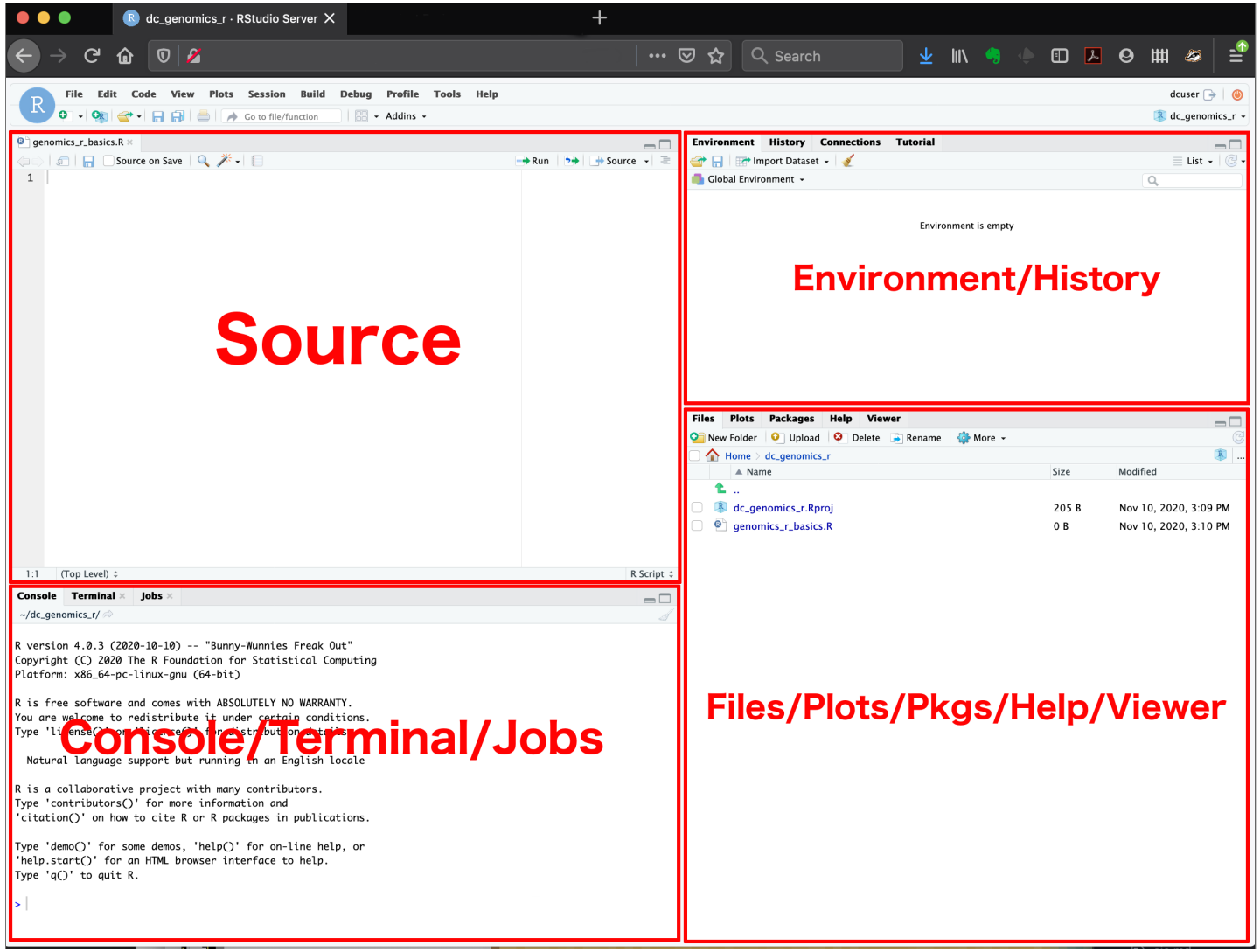
Elisa Pierfederici and Matthew Good
Digital Scholarship Center

3/13/23

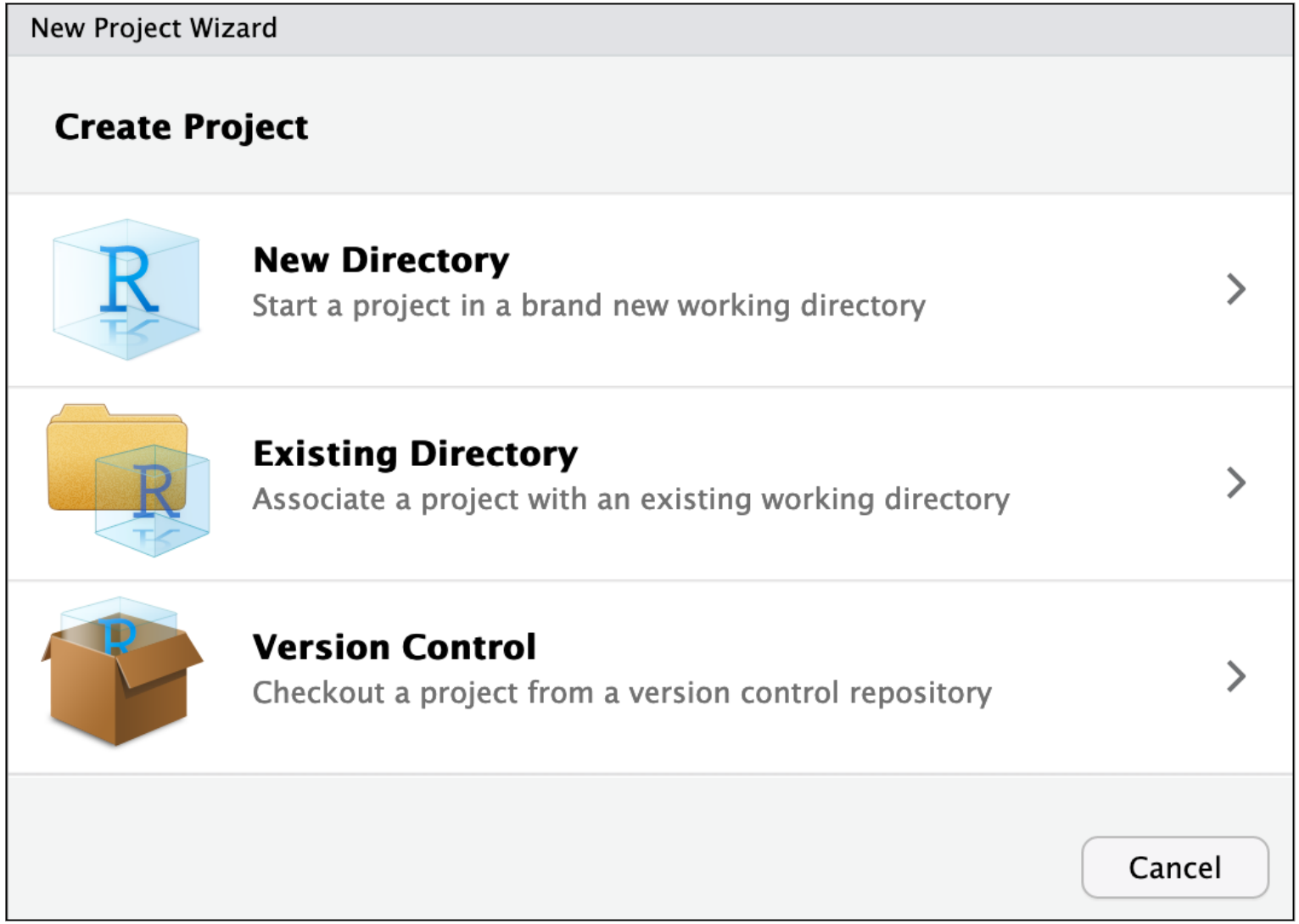
Data science project



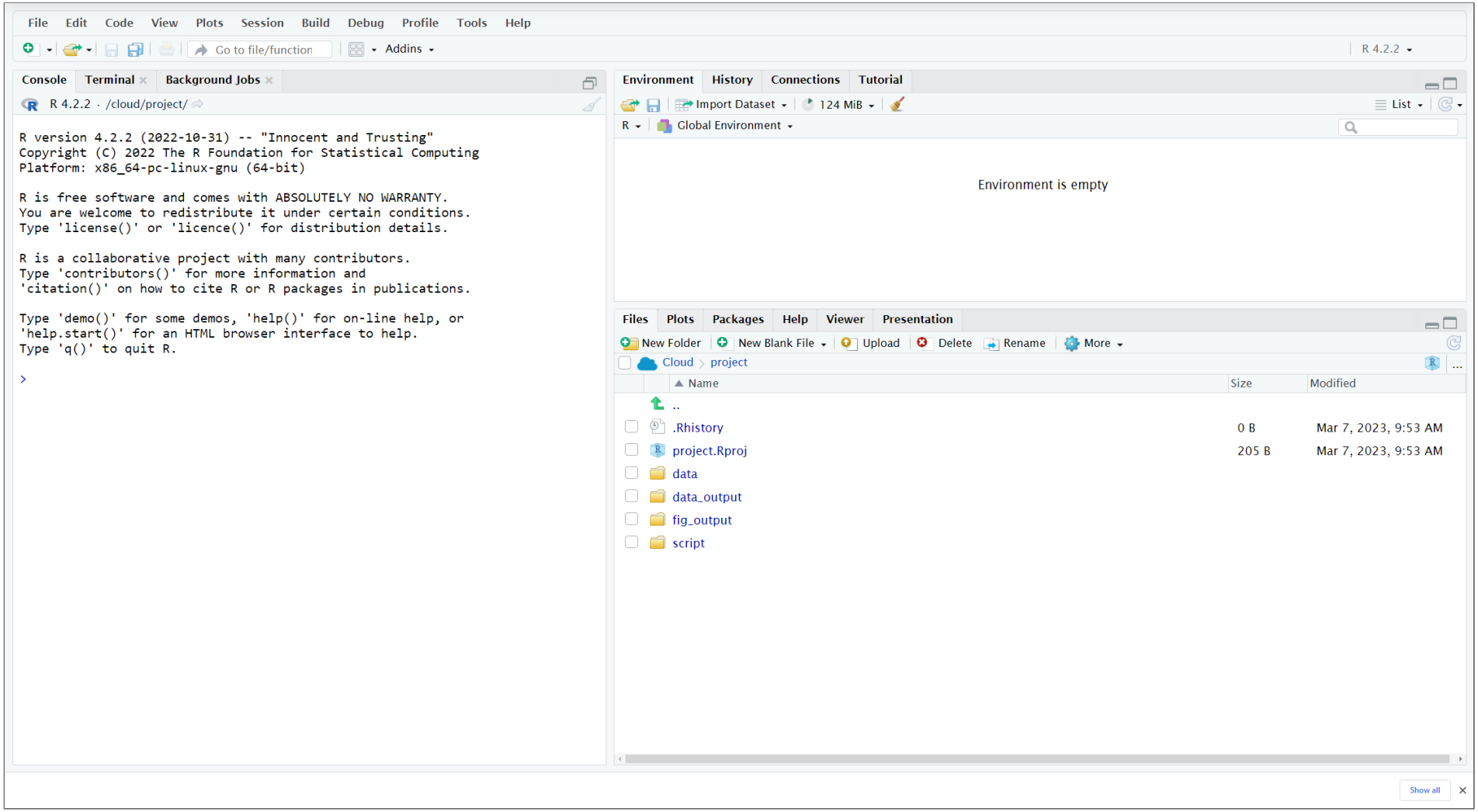
The RStudio interface



Create a project



Organizing your working directory



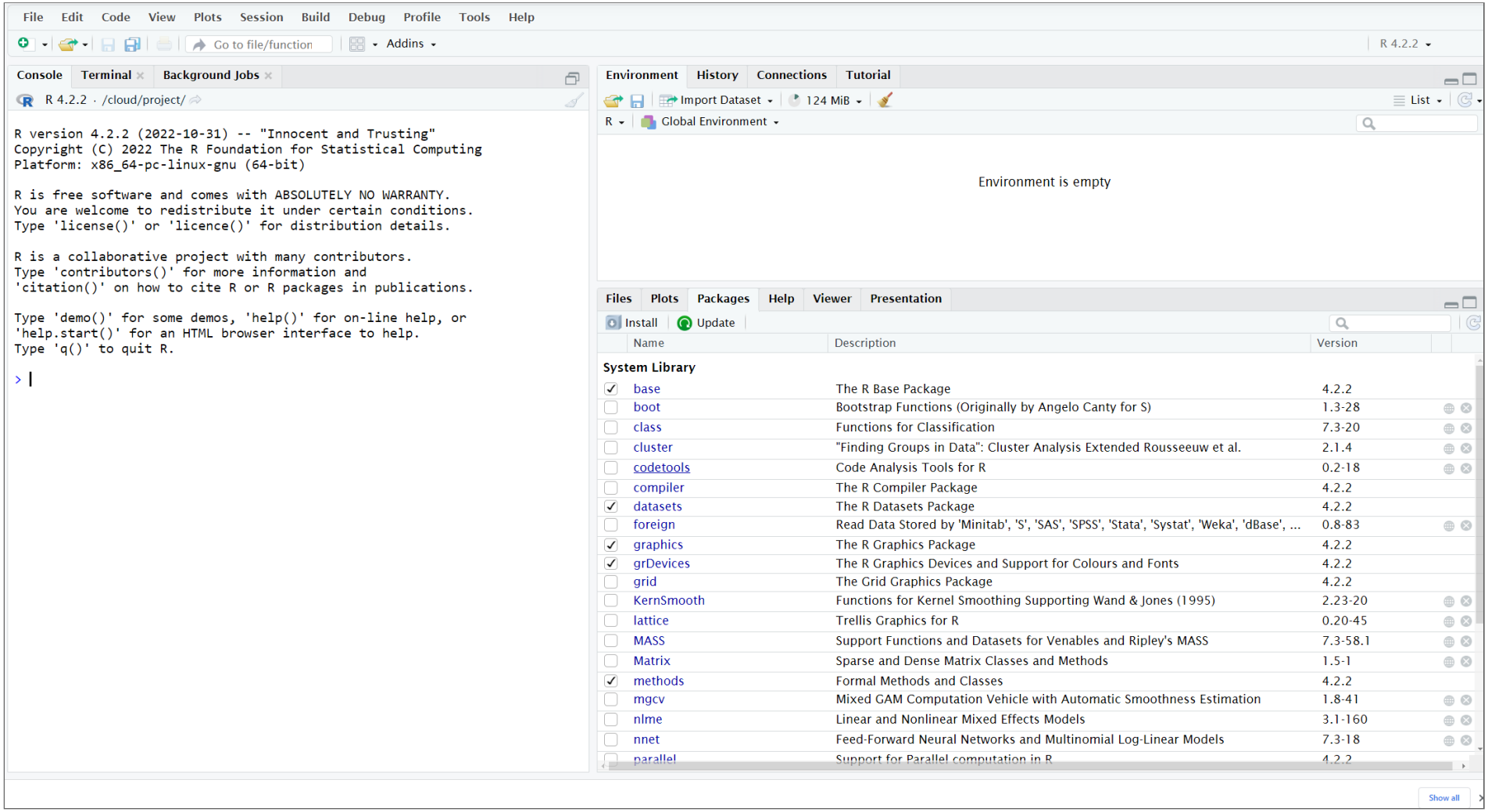
Create the first R script

1. Click the `File` menu and select `New File` and then `R Script`
2. Save your script by clicking the `save/disk icon` that is in the bar above the first line in the script editor

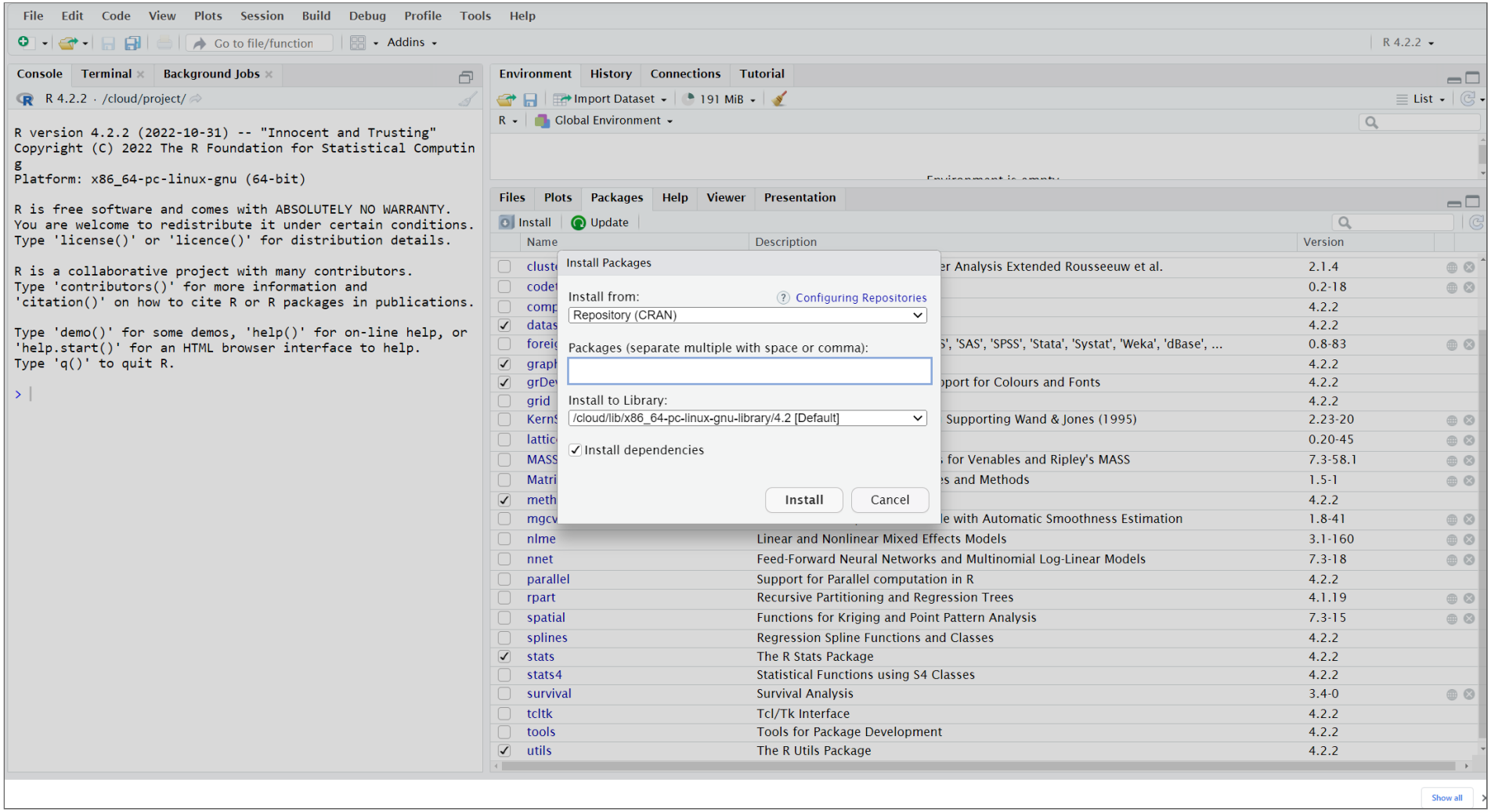
Downloading the data

1. Download the dataset called "`SAFI_clean.csv`" **https://ndownloader.figshare.com/files/11492171**
2. Place this downloaded file in the `data folder` you just created.

Installing packages



Installing packages



Shortcut - run a code

- PC: `Ctrl` + `Enter`
- Mac: `Cmd`+ `Return`

Creating an object - `assignment`

You can create new objects with `<-`:

```
1  area_hectares <- 1.0
```

Inspect an object

```
1  area_hectares
```

```
[1] 1
```

Descriptive names

- Object names must `start with a letter`
- Only contain letters, numbers, _ and `.`

Recomanded `snake_case` separate lowercase words with _

Descriptive names - examples

```r
this_is_a_really_long_name <- 2.5 # numeric
this_is_a_really_long_name
```

```
[1] 2.5
```

```r
x <- "hello world" # character
x
```

```
[1] "hello world"
```

```r
hh_members <- c(3, 7, 10, 6) # vector of numbers
hh_members
```

```
[1]  3  7 10  6
```

Data types & structure

Types of data

Type	Definition	Example
Integer	whole numbers from -inf to +inf	1L, -2L
Numeric/Double	numbers, fractions & decimals from -inf to + inf	7, 0.2, -5/2
Character/String	quoted strings of letters, numbers, and allowed symbols	“1”, “one”, “o_n_e”, “hello”
Logical	logical constant of True or False	`TRUE`, `FALSE`, `T`, `F`

Types of data

You can use **`typeof()`** to find out the type of value or object

```
1 typeof(1)
```

```
[1] "double"
```

```
1 typeof(TRUE)
```

```
[1] "logical"
```

```
1 typeof("one")
```

```
[1] "character"
```

Types of data

Type	Defintion	Example
NA	Missing value; technically represented as different types but displayed as `NA`	`NA`
NULL	The `NULL` object; an object that exists but is completely empty	`NULL`

Data structure

Vectors

Often, we are not working with individual values, but with multiple related values — vector of values!

We can create a vector of ordered numbers using the form

`starting_number:ending_numbers`

```
1 x <- 1:5
2 x
```

```
[1] 1 2 3 4 5
```

Lets look at the Environment pane in RStudio

Vectors

We can create a vector of any numbers we want using `c()`, which is a **function**.

```
# combine values into a vector and assign to an object names "x"
x <- c(2, 8.5, 1, 9)

# print x
x
```

```
[1] 2.0 8.5 1.0 9.0
```

Vectors

Vectors are just 1-dimensional sequences **of a single type of data**.

Note that vectors can also include strings or character values.

```
1 letters <- c("a", "b", "c", "d")
2 letters
```

```
[1] "a" "b" "c" "d"
```

Vectors

The general rule R uses is to set the vector to be the most "permissive" type necessary.
For example, what happens if we combine the vectors `x` (from earlier) and `letters` together?

```
1 mixed_vec <- c(x, letters)
2 mixed_vec
```

```
[1] "2"   "8.5" "1"   "9"   "a"   "b"   "c"   "d"
```

Vectors

```
1 mixed_vec
```

```
[1] "2"   "8.5" "1"   "9"   "a"   "b"   "c"   "d"
```

Notice the quotes? R turned all of our numbers into strings, since strings are more "permissive" than numbers.

```
1 typeof(mixed_vec)
```

```
[1] "character"
```

This is called **coercion**. R coerces a vector into whichever type will accommodate all of the values

Vectors

We can coerce `mixed_vec` to be numeric using **`as.numeric()`**, notice what happens to the character values

```
1  as.numeric(mixed_vec)
```

```
[1] 2.0 8.5 1.0 9.0  NA  NA  NA  NA
```

Help

RStudio

```
1  ?as.numeric
```

Google

1

Q1 Q2 Q3

Create an object called a that is just the letter “a” and an object x that is assigned the number 8. Add a to x. What happens?
Create a vector called b that is just the number 8 in quotes. Add b to x (from above). What happens?
Find some way to add b to x. (*Hint*: Don’t forget about coercion.)

Solution 1

Q1 | Q2 | Q3

```r
a <- "a"
x <- 8
## a + x
```

```r
b <- "8"
## b + x
```

```r
as.numeric(b) + x
```

```
[1] 16
```

Indexing vectors

How do we extract elements out of vectors?
This is called **indexing**, and it is frequently quite useful
There are a number of methods for indexing that are good to be familiar with

Indexing by position

Vectors can be indexed numerically, starting with 1 (not 0). We can extract specific elements from a vector by putting the index of their position inside brackets `[ ]`.

Indexing by position

Let's take a new vector `z` as an example:

```
1  z <- 6:10
```

Let's get just the first element of `z`:

```
1  z[1]
```

```
[1] 6
```

Get the first and third element by passing those indexes as a vector using `c()`.

```
1  z[c(1, 3)]
```

```
[1] 6 8
```

List

While vectors are useful for storing a single type of data, they're not well-suited for storing heterogeneous data. In other words, if we have different types of data we want to store together, we need a different data structure.

List

For example, let's say we want to store the year we're in a PhD program (a number), our name (a string), and our enrollment status (a logical) in a single object that preserves these different types. In this case, a vector won't work because it can only contain elements of the same type. Instead, we can use a list.
Lists are similar to vectors in that they're 1-dimensional, but they can store heterogeneous data. In other words, each element in a list can be a different type of data. This makes lists a more flexible data structure for storing complex or diverse data.

Creating Lists

We can create a list with the `list()` function

```
1 brendan <- list(4L, "Brendan Cullen", TRUE)
2 brendan
```

```
[[1]]
[1] 4

[[2]]
[1] "Brendan Cullen"

[[3]]
[1] TRUE
```

Creating Lists

And, we can give each element of the list a name to make it easier to keep track of them.

```
1 brendan <- list(year = 4L,
2                 name = "Brendan Cullen",
3                 enrollment = TRUE)
4 brendan
```

```
$year
[1] 4

$name
[1] "Brendan Cullen"

$enrollment
[1] TRUE
```

Notice that [[1]], [[2]], and [[3]], the element indices, have been replaced by the names year, name, and enrollment 👀

Creating Lists

You can see also see the names of a list by running `names()` on it

```
1 names(brendan)
```

```
[1] "year"       "name"       "enrollment"
```

Indexing Lists

If we want the actual object stored at the first position instead of a list containing that object, we have to use double-bracket indexing list[[i]]:

```
1  brendan[[1]]
```

```
[1] 4
```

Break

Data frames

A **data frame** is a common way of representing rectangular data -- collections of values that are each associated with a variable (column) and an observation (row). In other words, it has 2 dimensions.
A data frame is technically a special kind of list -- it can contain different kinds of data in different columns, but each column must be the same length.

Data frames

Import data frames

From *.csv*

1. Upload **`package tidyverse`**
2. Load the library

```
## load the tidyverse
library(tidyverse)
library(here)
```

Import data frames

From *.csv*

3. Import data frame and call it interview

```
interviews <- read_csv(here("data", "SAFI_clean.csv"), na = "NULL")
```

4. Inspect database

```
## inspect the data
interviews
## view(interviews)
## head(interviews)
```

Presentation of the SAFI Data

SAFI (Studying African Farmer-Led Irrigation) is a study looking at farming and irrigation methods in Tanzania and Mozambique. The survey data was collected through interviews conducted between November 2016 and June 2017. For this lesson, we will be using a subset of the available data. For information about the full teaching dataset used in other lessons in this workshop, see the **dataset description**.

Inspecting data frames

Size

- `dim(interviews)` - returns a vector with the number of rows as the first element, and the number of columns as the second element (the dimensions of the object)
- `nrow(interviews)` - returns the number of rows `ncol(interviews)` - returns the number of columns

Inspecting data frames

Content:

- `head(interviews)` - shows the first 6 rows `tail(interviews)` - shows the last 6 rows

Inspecting data frames

Summary:

- `str(interviews)` - structure of the object and information about the class, length and content of each column
- `summary(interviews)` - summary statistics for each column

Subsetting data frames

Let's get the first row and third column of `interviews` using numerical indexing

```
## first element in the first column
interviews[1, 1]
```

```
# A tibble: 1 × 1
  key_ID
   <dbl>
1      1
```

```
## first element in the 6th column of the tibble
interviews[1, 6]
```

```
# A tibble: 1 × 1
  respondent_wall_type
  <chr>
1 muddaub
```

Subsetting data frames

```
## first three elements in the 7th column of the tibble
interviews[1:3, 7]
```

```
# A tibble: 3 × 1
  rooms
  <dbl>
1     1
2     1
3     1
```

```
## equivalent to head_interviews <- head(interviews)
head_interviews <- interviews[1:6, ]
```

Negative Subsetting data frames

```r
# The whole tibble, except the first column
interviews[, -1]
```

```
# A tibble: 131 × 13
   village  interview_date      no_membrs years_…¹ respo…² rooms memb_…³ affec…⁴
   <chr>    <dttm>                  <dbl>    <dbl> <chr>   <dbl> <chr>   <chr>
 1 God      2016-11-17 00:00:00         3        4 muddaub     1 <NA>    <NA>
 2 God      2016-11-17 00:00:00         7        9 muddaub     1 yes     once
 3 God      2016-11-17 00:00:00        10       15 burntb…     1 <NA>    <NA>
 4 God      2016-11-17 00:00:00         7        6 burntb…     1 <NA>    <NA>
 5 God      2016-11-17 00:00:00         7       40 burntb…     1 <NA>    <NA>
 6 God      2016-11-17 00:00:00         3        3 muddaub     1 <NA>    <NA>
 7 God      2016-11-17 00:00:00         6       38 muddaub     1 no      never
 8 Chirodzo 2016-11-16 00:00:00        12       70 burntb…     3 yes     never
 9 Chirodzo 2016-11-16 00:00:00         8        6 burntb…     1 no      never
10 Chirodzo 2016-12-16 00:00:00        12       23 burntb…     5 no      never
# … with 121 more rows, 5 more variables: liv_count <dbl>, items_owned <chr>,
#   no_meals <dbl>, months_lack_food <chr>, instanceID <chr>, and abbreviated
#   variable names ¹years_liv, ²respondent_wall_type, ³memb_assoc,
```

2

Q1 Q2

Create a tibble (interviews_100) containing only the data in row 100 of the interviews dataset.
Notice how nrow() gave you the number of rows in the tibble?

- Use that number to pull out just that last row in the tibble.
- Compare that with what you see as the last row using tail() to make sure it’s meeting expectations.

Solution 2

Q1 | Q2

```r
## 1.
interviews_100 <- interviews[100, ]
interviews_100
```

```
# A tibble: 1 × 14
  key_ID village interview_date      no_membrs years_liv respond…¹ rooms memb_…²
   <dbl> <chr>   <dttm>                   <dbl>     <dbl> <chr>     <dbl> <chr>
1     80 Ruaca   2017-04-28 00:00:00         5        12 muddaub       1 no
# … with 6 more variables: affect_conflicts <chr>, liv_count <dbl>,
#   items_owned <chr>, no_meals <dbl>, months_lack_food <chr>,
#   instanceID <chr>, and abbreviated variable names ¹respondent_wall_type,
#   ²memb_assoc
```

```r
## 2.
# Saving `n_rows` to improve readability and reduce duplication
n_rows <- nrow(interviews)
interviews_last <- interviews[n_rows, ]
```

Grammar of data manipulation

Learning `dplyr`

`select()` picks variables based on their names.
`filter()` picks cases based on their values.
`mutate()` - adds or alters variables that are functions of existing variables
`summarise()` reduces multiple values down to a single summary.
`arrange()` changes the ordering of the rows.

`filter()` - subsetting rows

`select()` - reduce columns

Reducing the number of columns (or rearranging columns)

Select columns

```
# to select columns throughout the dataframe
select(interviews, village, no_membrs, months_lack_food)
# to do the same thing with subsetting
interviews[c("village","no_membrs","months_lack_food")]
# to select a series of connected columns
select(interviews, village:respondent_wall_type)
```

Filtering rows

```
# filters observations where village name is "Chirodzo"
filter(interviews, village == "Chirodzo")
```

Filtering rows

```
# filters observations with "and" operator (comma)
# output dataframe satisfies ALL specified conditions
filter(interviews, village == "Chirodzo",
                   rooms > 1,
                   no_meals > 2)
```

Filtering rows

```
# filters observations with "&" logical operator
# output dataframe satisfies ALL specified conditions
filter(interviews, village == "Chirodzo" &
                   rooms > 1 &
                   no_meals > 2)
```

Shortcut - the pipe

- PC: `Ctrl` + `Shift` + `M`
- Mac: `Cmd` + `Shift` + `M`

The pipe

```r
# standard
interviews2 <- filter(interviews, village == "Chirodzo")
interviews_ch <- select(interviews2, village:respondent_wall_type)

# piped
interviews_ch <- interviews %>%
    filter(village == "Chirodzo") %>%
    select(village:respondent_wall_type)

interviews_ch
```

3

Using pipes, subset the interviews data to include interviews where respondents were members of an irrigation association (memb_assoc) and retain only the columns affect_conflicts, liv_count, and no_meals.

Solution 3

```
interviews %>%
    filter(memb_assoc == "yes") %>%
    select(affect_conflicts, liv_count, no_meals)
```

`Mutate()` - add column

create new columns based on the values in existing columns
Example: we are interest in the avg number of people per room

```
interviews %>%
    mutate(people_per_room = no_membrs / rooms)
```

`Mutate()` - add column

We may be interested in investigating whether being a member of an irrigation association had any effect on the ratio of household members to rooms. To look at this relationship, we will first remove data from our dataset where the respondent didn't answer the question of whether they were a member of an irrigation association. These cases are recorded as "NULL" in the dataset.

```
interviews %>%
    filter(!is.na(memb_assoc)) %>%
    mutate(people_per_room = no_membrs / rooms)
```

`Mutate()` - add column

- `is.na()` returns a value of `TRUE` (because the memb_assoc is missing)
- the `!` symbol negates this and says we only want values of `FALSE`, where memb_assoc is not missing.

📝4

Create a new dataframe from the `interviews` data that meets the following criteria: contains only the `village` column and a new column called `total_meals` containing a value that is equal to the total number of meals served in the household per day on average (`no_membrs` times `no_meals`). Only the rows where `total_meals` is greater than 20 should be shown in the final dataframe.

Hint: think about how the commands should be ordered to produce this data frame!

Solution 4

```
interviews_total_meals <- interviews %>%
    mutate(total_meals = no_membrs * no_meals) %>%
    filter(total_meals > 20) %>%
    select(village, total_meals)
```

Count() - counting

If we wanted to count the number of rows of data for each village

```
interviews %>%
    count(village)
```

```
# A tibble: 3 × 2
  village      n
  <chr>    <int>
1 Chirodzo    39
2 God         43
3 Ruaca       49
```

```
interviews %>%
    count(village, sort = TRUE) # to get results in decreasing order
```

```
# A tibble: 3 × 2
  village      n
  <chr>    <int>
1 Ruaca       49
2 God         43
3 Chirodzo    39
```

Summarize() - summary statistic

- `group_by()` collapses each group into a single-row summary of that group.

```
interviews %>%
    group_by(village)
```

Summarize() - summary statistic

- takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics.

```
interviews %>%
    group_by(village) %>%
    summarize(mean_no_membrs = mean(no_membrs))
```

```
interviews %>%
    group_by(village, memb_assoc) %>% # group by multiple col
    summarize(mean_no_membrs = mean(no_membrs))
```