Foundations of Data Science for Biologists

R: importing data, data types, and indexing

BIO 724D

11-SEP-2023

Instructors: Jesse Granger, Greg Wray, and Paul Magwene

Importing data

Packages

What is a package?

- Packages are extensions that contain functions, code and sample data in a standardized format
- Sort of like DLC's or Mods for video games

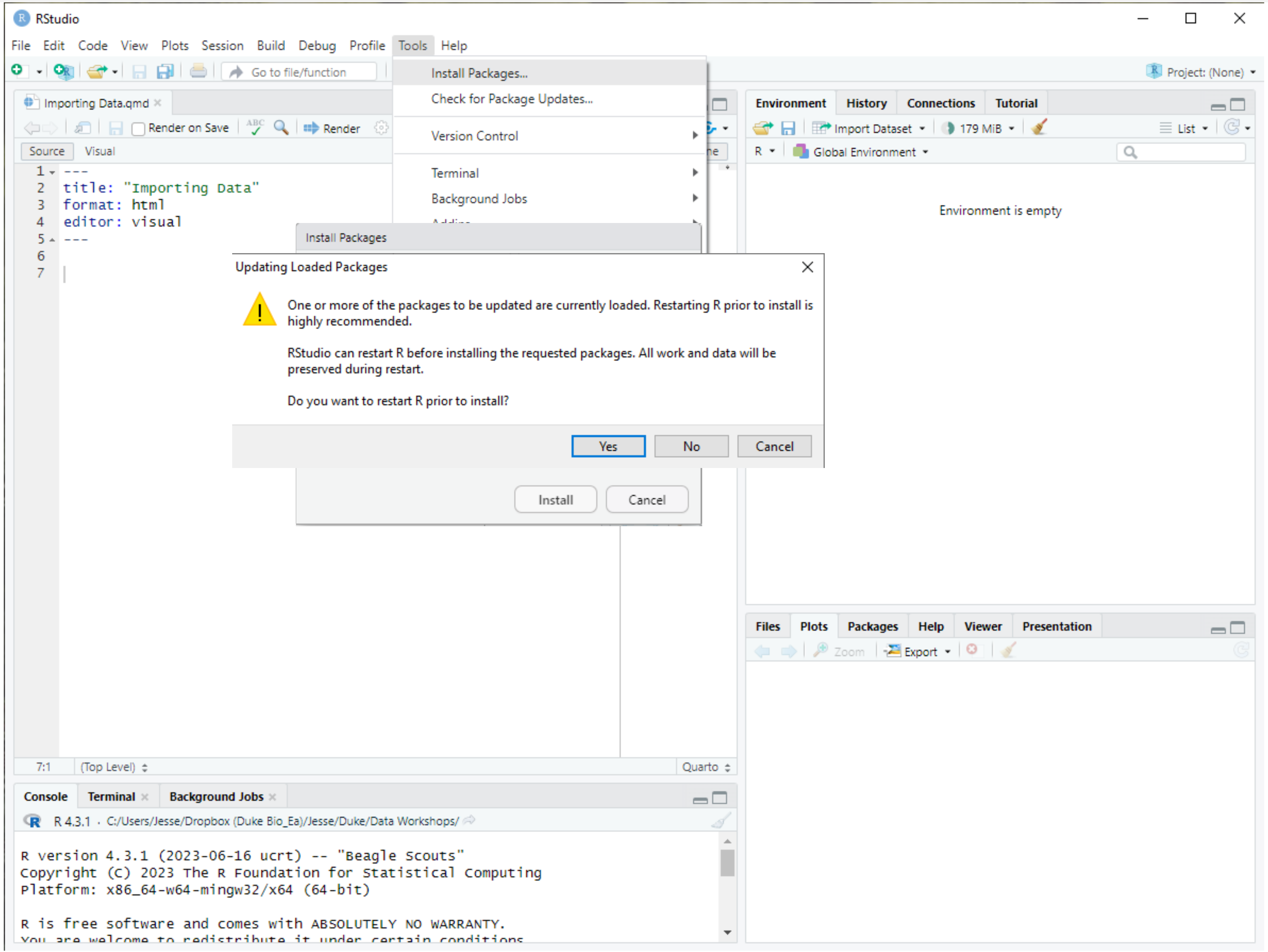
Where do you get packages from?

- CRAN
- Github, or other sources less common

Getting a package into R

- Step one: INSTALL (you only ever have to do this once)
- Step two: Use the package by calling it from the library

Installing Packages



```
---
title: "Importing Data"
format: html
editor: visual
---
```

Updating Loaded Packages

One or more of the packages to be updated are currently loaded. Restarting R prior to install is highly recommended.

RStudio can restart R before installing the requested packages. All work and data will be preserved during restart.

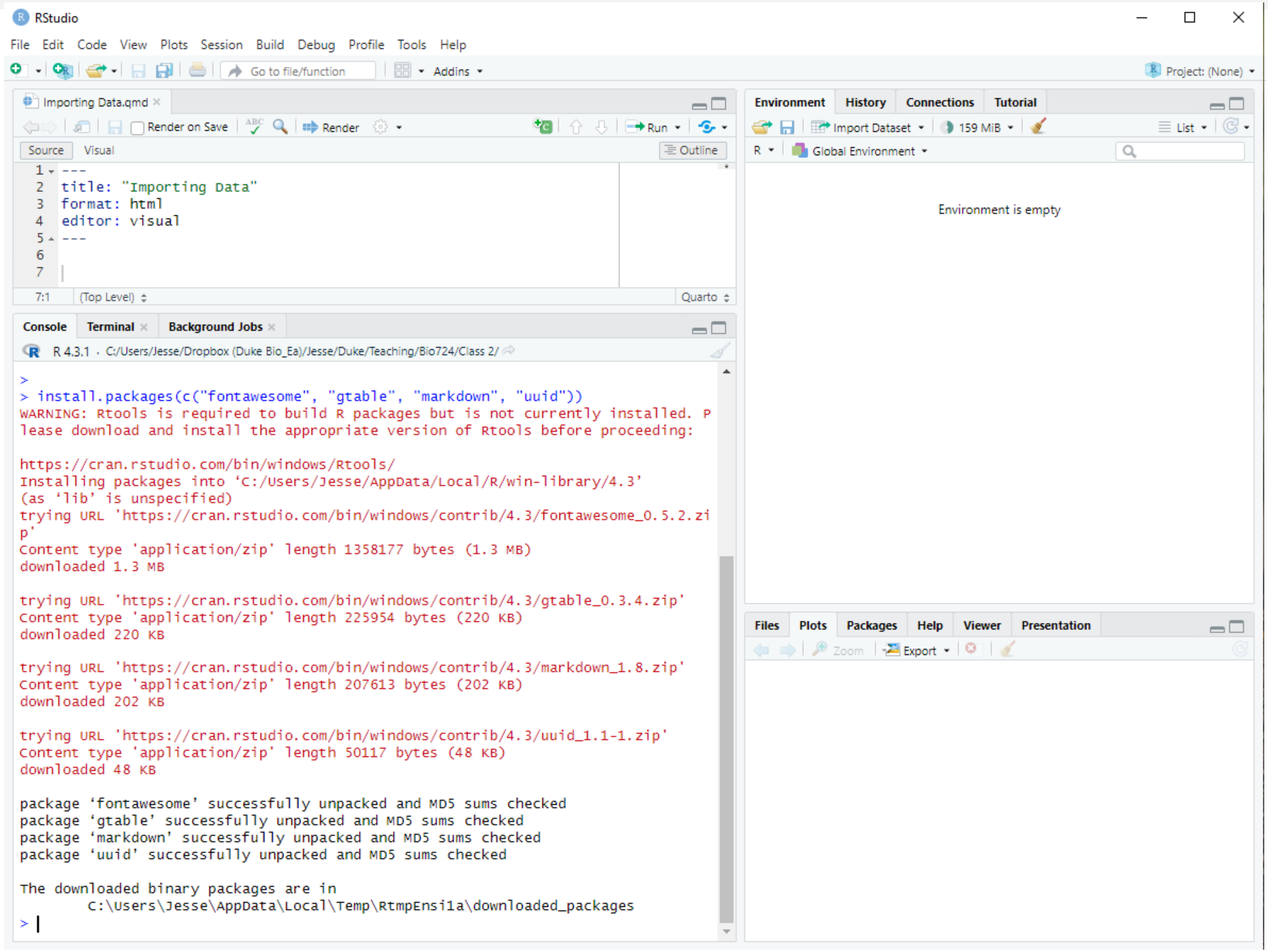
Do you want to restart R prior to install?

Yes No Cancel

```
R version 4.3.1 (2023-06-16 ucrt) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
```

```
Install.packages(readxl)
```

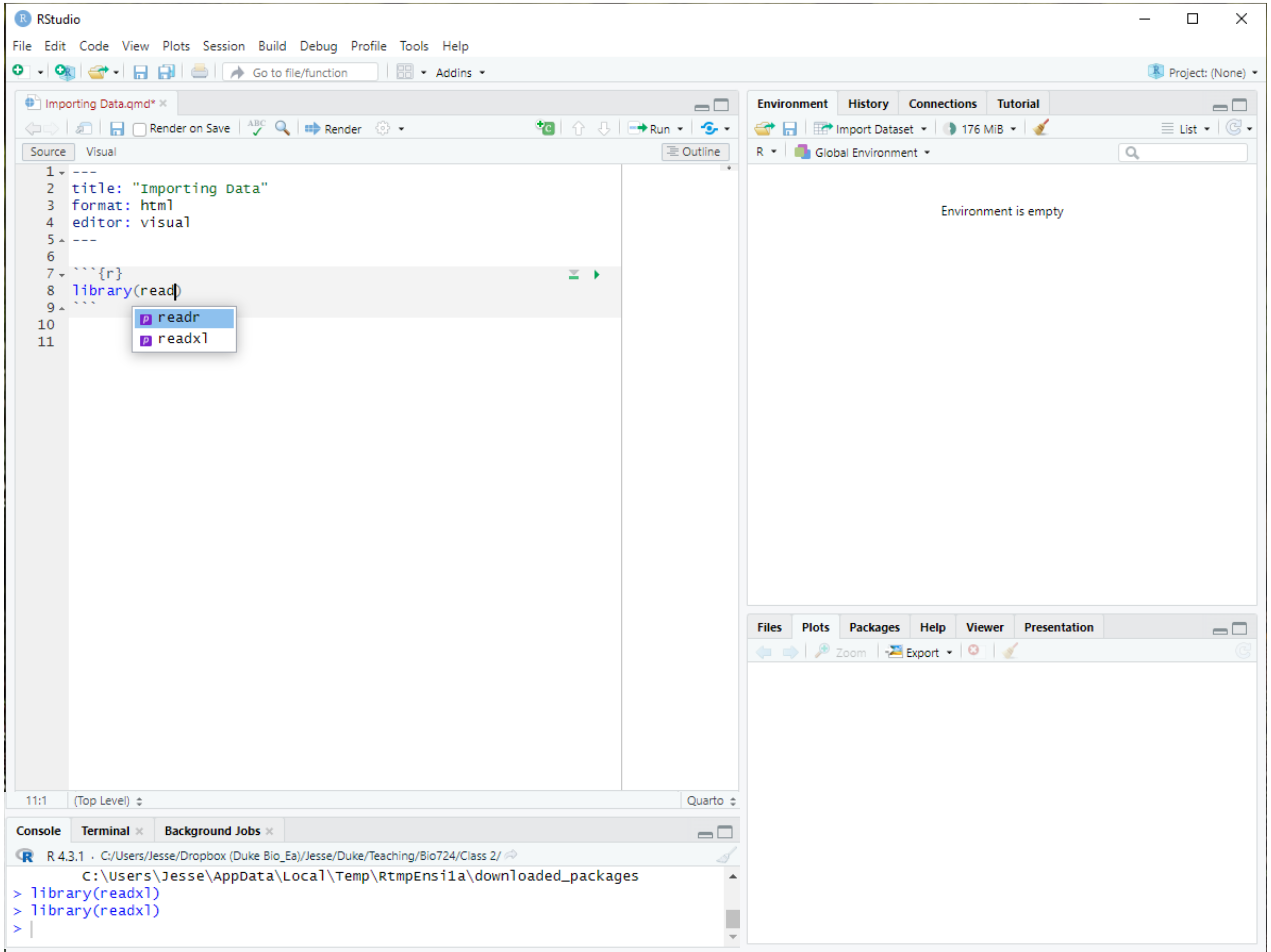
Side Note—Updating Packages



```
---
title: "Importing Data"
format: html
editor: visual
---

```

```
>
> install.packages(c("fontawesome", "gtable", "markdown", "uuid"))
WARNING: Rtools is required to build R packages but is not currently installed. P
lease download and install the appropriate version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
Installing packages into ‘C:/Users/Jesse/AppData/Local/R/win-library/4.3’
(as ‘lib’ is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.3/fontawesome_0.5.2.zi
p'
Content type 'application/zip' length 1358177 bytes (1.3 MB)
downloaded 1.3 MB

trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.3/gtable_0.3.4.zip'
Content type 'application/zip' length 225954 bytes (220 KB)
downloaded 220 KB

trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.3/markdown_1.8.zip'
Content type 'application/zip' length 207613 bytes (202 KB)
downloaded 202 KB

trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.3/uuid_1.1-1.zip'
Content type 'application/zip' length 50117 bytes (48 KB)
downloaded 48 KB

package ‘fontawesome’ successfully unpacked and MD5 sums checked
package ‘gtable’ successfully unpacked and MD5 sums checked
package ‘markdown’ successfully unpacked and MD5 sums checked
package ‘uuid’ successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Jesse\AppData\Local\Temp\RtmpEnsi1a\downloaded_packages
> |
```

Loading Packages



~~~
---
title: "Importing Data"
format: html
editor: visual
---

```{r}
library(read)
```
~~~

```
R 4.3.1 · C:/Users/Jesse/Dropbox (Duke Bio_Ea)/Jesse/Duke/Teaching/Bio724/Class 2/
    C:\Users\Jesse\AppData\Local\Temp\RtmpEnsi1a\downloaded_packages
> library(readxl)
> library(readxl)
>
```

In Class Exercise

Install the package readr and then call it from the library

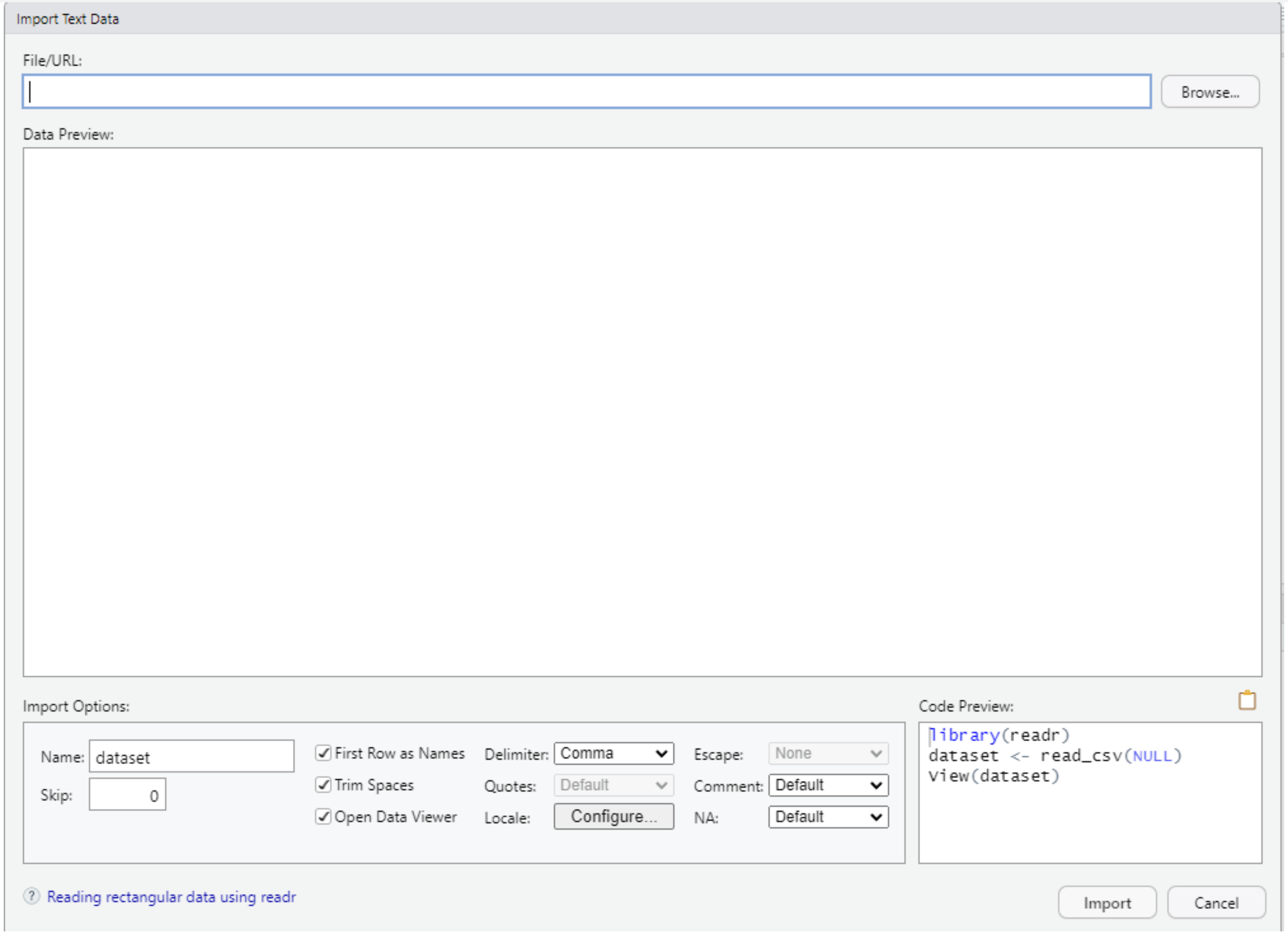
The most common types of data

- .csv
- .tsv
- .txt
- .xlsx

Things to pay attention to:

- Delineators/Separators
- Na.strings
- Headers/column names and rownames

Importing Data—The Cheater Way



Name: dataset
Skip: 0
✓ First Row as Names
✓ Trim Spaces
✓ Open Data Viewer
Delimiter: Comma
Quotes: Default
Locale: Configure...
Escape: None
Comment: Default
NA: Default

```
library(readr)
dataset <- read_csv(NULL)
View(dataset)
```

Data types and assignment

Data types

Computers work with 0s and 1s — but you want to work with numbers, names, dates, etc.

- **Data types** instruct programs how to interpret and process different kinds of data
- For instance, division is useful for numbers but illogical for dates or truth values
- Basic data types are number, string, logical, etc.

R has an extensive set of rules for each data type:

- What **values** are allowed (e.g., 0 and 1 for logical)
- What **operations** are allowed (e.g., division for integers but not dates or strings)
- How to **display** data in human-readable form (e.g., 00110010 00110111 as 27)

When R stores values in memory or a file, two things happen

- **Encoding**: human-readable values are converted into 0s and 1s (data)
- **Typing**: information is stored regarding data type and other properties (metadata)

Data types and assignment in R

A variable’s data type is inferred by R at the time of assignment (**soft typing**)

- `my_var <- 3.14159` interprets and assigns value as numeric
- Some languages require variable type to be explicitly declared (e.g., C++)

A variable’s value can be updated (**re-bound**) at any time, including changing data type

- `my_var <- “hello world”` interprets and assigns value as character
- Re-binding is silent (no warning or error message), because it is very useful

A variable only exists for the current session

- To re-create variables in a later session, add the appropriate steps to your script
- Alternatively, save the data in a file, and read its contents in a later session

Naming variables

R has some simple rules for naming variables:

- Must start with a letter *or* . (dot) immediately followed by a letter
- May include: letters, numbers, underscore, dot, standard keyboard symbols
- May not include spaces (there is a work-around, but spaces are usually a bad idea)
- Case-sensitive
- Can be arbitrarily long
- Cannot be a **reserved word**; type `help(reserved)` or `?reserved`

Best practices

- When writing programs, favor descriptive, long names over simple, short ones
- Avoid relying on case and using symbols (other than underscore and dot)
- Avoid naming variables with the names of functions (although this is allowed!)

Atomic data types in R

Four atomic (most basic) data types are very commonly used:

- **Numeric**: real numbers; double-precision floating point by default
- **Integer**: whole numbers
- **Logical**: `TRUE`, `FALSE` (called **Boolean** in some languages), and `NA`
- **Character**: strings composed of letters, numerals, symbols, and whitespace

Two additional atomic data types are available but rarely needed:

- **Complex**: imaginary numbers with values like `2+3i`, where $i^2 = -1$
- **Raw**: bytes with no implied meaning

Atomic data types are always **vectors** (vector = values of same type in a specific order)

- If you assign just one value, the result is a vector of length 1
- You can assign multiple values at once using the concatenate function, `c()`

Data objects

The process of assignment creates a package of information called a **data object**

The variable name and associated values are stored together in memory

Less obviously, metadata are also stored: data type, length, names, and often more

You can learn about a data object in several ways, including:

`my_var`	returns current value(s)
`typeof(my_var)`	returns the object's specific data type
`class(my_var)`	returns the object's more general data type or structure
`length(my_var)`	returns the number of items in the data object
`str(my_var)`	returns a description of the structure of a data object
`attributes(my_var)`	returns the non-standard metadata of a data object
`View(my_var)`	displays all the data in a scrollable window (RStudio only)

Data structures in R

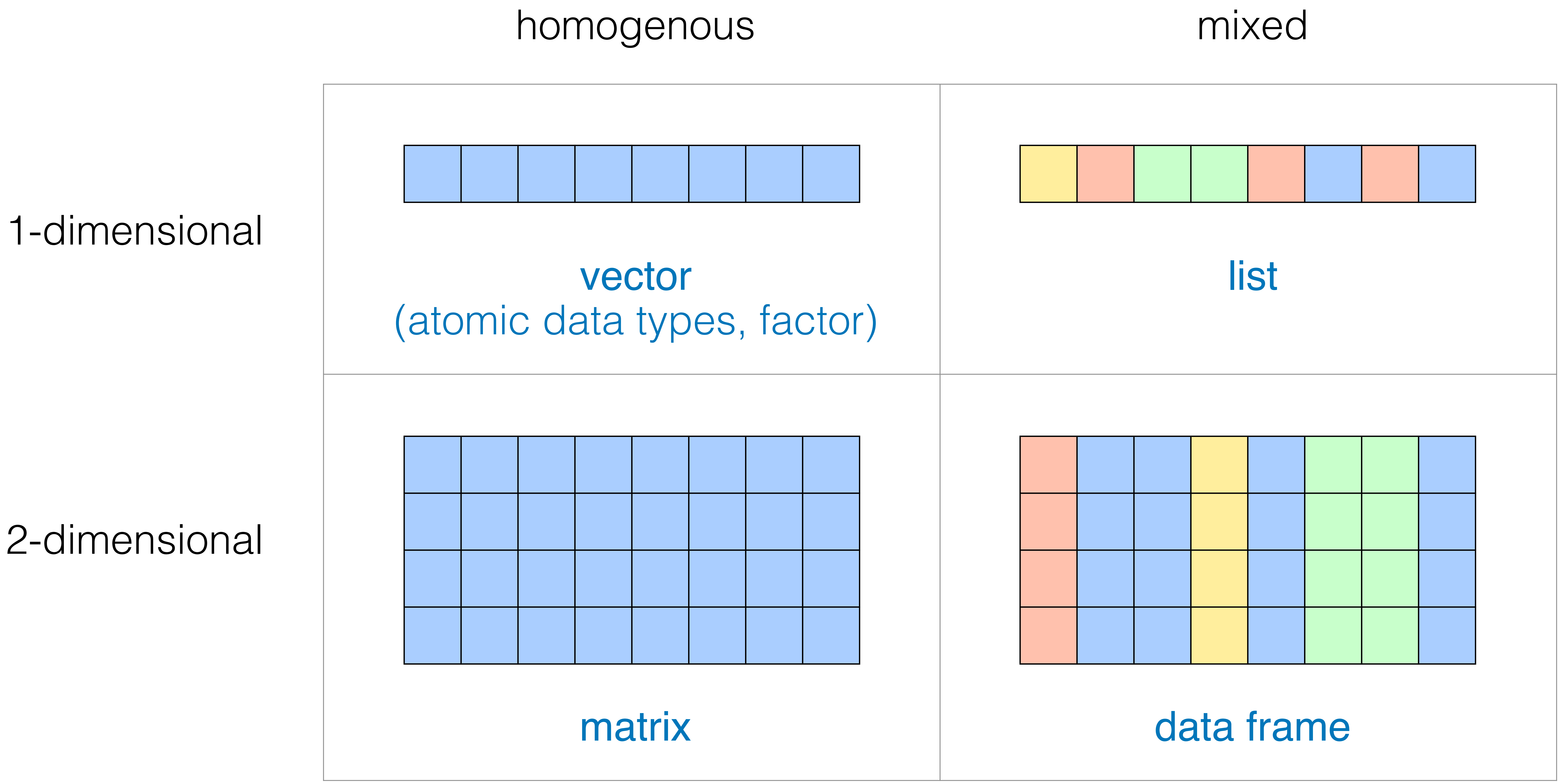
A **data structure** is a data object constructed from more basic data types

- Allows you to organize data in a consistent way for processing by 3rd-party code
- Allows you to attach useful metadata, such as column labels or factor levels
- Internally, data structures are organized to optimize processing (e.g., sorting)

Three common data structures in R are:

- **List**: a vector that can contain different kinds of items (even lists! even functions!)
- **Data frame**: similar to a spreadsheet (more formally, a list of equal-length vectors)
- **Factor**: an integer vector with string labels and special functionality

Taxonomy of basic data types in R



Converting between data types

It is often possible and useful to convert between data types (called **coercion** in R)

- Must be a homogenous data type (vector, matrix, or column in a data frame)
- Must make logical sense (e.g., "2" can be coerced to integer but "kangaroo" cannot)

To coerce, use `as.integer()`, `as.logical()`, `as.character()`, etc.

Coercion rules to be aware of:

Numeric to integer	truncates any decimal values (does not round!)
Numeric to logical	`0` becomes `FALSE`; non-zero values become `TRUE`
Logical to numeric	`TRUE` becomes `1`, `FALSE` becomes `0`
Numeric to character	numerals and symbols become characters
Character to numeric	must be a formatted number (`-`, `+` and `.` allowed)

And many more; check documentation to avoid unexpected results!

Testing for data type

R allows you to re-assign different data types to the same variable name

- This make programming simpler and code easier to read
- However, it can create problems if you forget the data type of a variable

```
my_var <- 42L                        # assign an integer value
. . .                                # some other code
my_var <- “the meaning of life”      # assign an equivalent string (Adams 1979)*
new_var <- my_var * 2                # throws an error because value is character
```

It’s good practice to test for data type before processing unfamiliar data

- Use `is.integer()`, `is.logical()`, `is.character()`, etc. to evaluate data type
- E.g., `is.character(my_var)`, returns TRUE after the code above runs

*Adams, D (1979) *The Hitchhikers’s Guide to the Galaxy*. Crown, New York.

Missing values

R provides three special values that represent missing, invalid, or undefined information

`NA`	a missing value; acronym for not available
`NaN`	an invalid mathematical result (e.g., `0/0`); acronym for not a number
`NULL`	a value that is undefined (e.g. vector of length `0`)

Points to remember:

Do not use quotes: `‘NA’` is interpreted a character value

Do not use in mathematical operations: `my_var + NA` substitutes every item with `NA`

Do not use in logical tests: `my_var == NA` returns `NA`

To identify missing values:

`is.na(my_vec)`	returns a logical vector with NAs FALSE, all others TRUE
`which(is.na(my_vec))`	returns the position(s) of any NAs in the vector

Indexing vectors

Indexing allows you to access specific values within a vector:

`my_obj`	12	7	23	0	8	-2	4	5	← values
	1	2	3	4	5	6	7	8	← indexes

The ordinal position of a value is it's **index**

R uses **1-based** indexing (unlike many other programming languages)

Refer to specific values using square brackets:

`my_result <- my_obj[3]` assigns the value `23` to `my_result`

Refer to ranges using a colon:

`my_result <- my_obj[2:4]` assigns the vector `(7, 23, 0)` to `my_result`

Basics of assignment

The basic form of assignment is:

`my_obj <- 7`	read as: “my_object gets 7”
`my_obj <- c(7, 14, 21)`	creates a numeric vector of length 3

Other valid forms of assignment:

`7 -> my_obj`	sometimes more readable
`my_obj <- 7 -> other_obj`	assigns 7 to 2 different variables at once
`A <- B <- C <- 7`	assigns 7 to 3 different variables at once
`my_obj = 7`	alternative assignment operator
`assign(my_obj, 7)`	using a function works, too (but awkward!)

Use a **shortcut** for the assignment operator: alt+minus (Win) / opt+minus (Mac)

Assignment has many uses

Store the result of an operation:

`my_var <- 1 + 2`	evaluates RHS and assigns result to LHS
`my_vec <- old_vec * 3`	multiples each element by 3 during assignment

Create a new data object:

`my_vec <- c(1:10)`	creates a numeric vector containing values 1-10
`my_list <- list(1, “a”)`	creates a list containing values 1 and “a”

Update 1 or more values in an existing data object:

`my_vec[10] <- 42`	changes the value of item 10 to 42
`mvec[1:length(mvec)] <- 42`	replaces every value with 42, preserving length
`my_vec <- 42`	re-binds my_vec to a single value, length = 1
`my_vec[is.na(my_vec)] <- 0`	replaces NAs with 0; other values unchanged

Assignment has many uses, *continued*

Add items to an existing data object:

`my_vec[11] <- 300`	adds 1 item to a vector containing 10 items
`my_vec[12:15] <- c(1,3,4)`	adds 3 items to a vector containing 11 items

Delete items from an existing data object:

`my_vec <- my_vec[c(1,3,6)]`	removes all items except 1, 3, and 6
`my_vec <- my_vec[1:3]`	removes all items except the first 3

Copies an existing vector:

`new_vec <- old_vec`	copies values from one vector to another

Create a logical vector to use for subsetting or counting:

`logic_vec <- age_vec < 3`	assigns TRUE and FALSE values accordingly
`logic_vec <- is.na(my_vec)`	assigns TRUE and FALSE values accordingly

Lists, data frames, and indexing

Review: Vectors

Structure

- Homogeneous – all items in a vector are of the same type
- Ordered – each item has a position in the vector

Indexing

- For a vector of length n, the indices are $1 \ldots n$ (1-indexing)
- Indexing occurs using single brackets `[]`

```r
x <- c(2, 4, 6, -99, NA)
x[1]
x[length(x)]  # robust way to get last element
```

Question: What is the type of the “NA” item in the vector example above?

Data Frames

Data Frames represent tabular data. You can think of the columns of a data frame as an ordered collection of vectors.

Columns

- Columns of a data frame represent variables
- Columns must have names
- Every item in a given column is of the same data type
- Each column can have a different data type
- Each column must be of the same number of rows

Rows

- Rows represent observations/entities
- Each row is of the same length
- A row is heterogeneous collection – a data frame with a single observation

Constructing a data frame from scratch

```r
name <- c("Paul", "Maira", "Peter", "Beatriz")
grade <- c("C", "A", "B", "A-")
age <- c(40, 12, 17, 52)

example.df <- data.frame(name = name,
                         age = age,
                         grade = grade)
example.df
```

```
     name age grade
1    Paul  40     C
2   Maira  12     A
3   Peter  17     B
4 Beatriz  52    A-
```

Data frames: Detail

Shape

```r
dim(example.df)   # number of rows and columns
nrow(example.df)  # number of rows
ncol(example.df)  # number of columns
```

Column names

```r
names(example.df)
```

Data frames: Indexing by position

- Every element in a data frame is indexed by a row and a column position

```
example.df[3, 2]  # row, column
```

- Get a single column by integer position

```
example.df[2]
```

- Get multiple columns using a vector of indices

```
example.df[c(1,3)]
```

- Get a single row by integer position (not comma)

```
example.df[2,]  # row 2
```

- Multiple rows uing vector of indices

```
example.df[c(1,3), ]  # note comma
```

Data frames: Indexing rows and columns simultaneously

You can simultaneously index both rows and columns:

```r
example.df[c(1,3), c("name", "age")]
```

Data frames: Column name indexing

Indexing columns by name:

```r
example.df["grade"]
```

You can get multiple columns at a time by indexing with a vector of column names

```r
example.df[c("name","grade")]
```

`$` operator

The `$` operator followed by the name of a column **returns a vector** representing the values in the corresponding data frame column:

```
example.df$grade
```

- Note that when using the `$` operator you don't have to put the name of the column in quotes unless there are spaces in the name

Double bracket indexing

The columns of a data frame can be accessed by double bracket indexing:

```
example.df[[1]]
```

Like the `$` operator this returns a vector.

Boolean indexing

Both vectors and lists can be “Boolean indexed” – given an indexing vector of logical (`TRUE`/`FALSE`) values, Boolean indexing returns all the elements where the indexing vector is `TRUE`

- Vector example

```
x <- c(1, 2, 3, 4)
x[c(TRUE, FALSE, FALSE, TRUE)]
```

```
[1] 1 4
```

- Data frame example

```
example.df[c(TRUE,FALSE,TRUE,FALSE), ]
```

```
   name age grade
1  Paul  40     C
3 Peter  17     B
```

Boolean indexing, continued

Boolean indexing is often used to filter or subset data

- Example: subsetting the rows of a data frame

```
# get all rows of data frame where persons age > 18
is.adult <- example.df$age > 18
is.adult
```

```
[1]  TRUE FALSE FALSE  TRUE
```

```
example.df[is.adult, ]
```

```
     name age grade
1    Paul  40     C
4 Beatriz  52    A-
```

Usually we'd write the above example like so:

```
example.df[example.df$age > 18, ]
```

We'll see a cleaner syntax and more example of Boolean indexing and filtering when we introduce the dplyr package

Lists

Lists are the most flexible built-in data structure in R.

- Unlike vectors and data frames which have constraints on what they contain and the size of the respective elements, lists can contain arbitrary objects of any type and size (even other lists)

```r
bob <- list('Bob', 16, 27707)

selena <- list('Selena', 'Montgomery', 17, 91324)

people <- list(bob, selena)

people
```

List elements can have names

```r
bob <- list(first_name="Bob",
            last_name="Gimli",
            age=16,
            zip=27707)
```

The names of list elements can be accessed with the `names()` function similar to the columns in a data frame

```r
names(bob)
```

Indexing Lists

- Single brackets always **return a list** containing the element at index `i`

```
bob[1]
typeof(bob[1])
```

- Use double brackets (`[[]]`)to return the element at index `i`

```
bob[[1]]
typeof(bob[[1]])
```

- If the list has named objects they can be accessed via the `$` operator

```
bob$last_name
```

- String indexing also works with lists

```
bob[c("last_name", "first_name")]
```

Question: What happens when you index with an integer index or a name that doesn't exist?

Functions often return lists

Many R functions that need to return multiple values of different types will return lists (or things that act like lists).

Example

```r
# draws a histogram but also return a list object
# with useful information about what was drawn
h <- hist(rnorm(100))
```

```r
names(h)
## [1] "breaks"   "counts"   "density"  "mids"     "xname"    "equidist"

# get break points for each bin and respective counts
h$breaks
## [1] -3 -2 -1  0  1  2  3
h$counts
## [1]  2 15 28 36 17  2
```

Setting values in data structures using indexing

The items in vectors, data frames, and lists can all be set or changed using the indexing operations described previously.

- Vector example

```
v1 <- c(2, 4, 6, 8)
v1
## [1] 2 4 6 8
v1[3] <- -99
v1
## [1]   2   4 -99   8
```

Setting values in data structures using indexing, cont.

- Data frame example:

```
example.df
##      name age grade
## 1    Paul  40     C
## 2   Maira  12     A
## 3   Peter  17     B
## 4 Beatriz  52    A-
```

```
example.df$middle_initial <- c("M", "M", "B", "S")
example.df
##      name age grade middle_initial
## 1    Paul  40     C              M
## 2   Maira  12     A              M
## 3   Peter  17     B              B
## 4 Beatriz  52    A-              S
```