# Introduction to Data Analysis in R

BIOL90041

Week 2

Dr. Kristoffer Wild

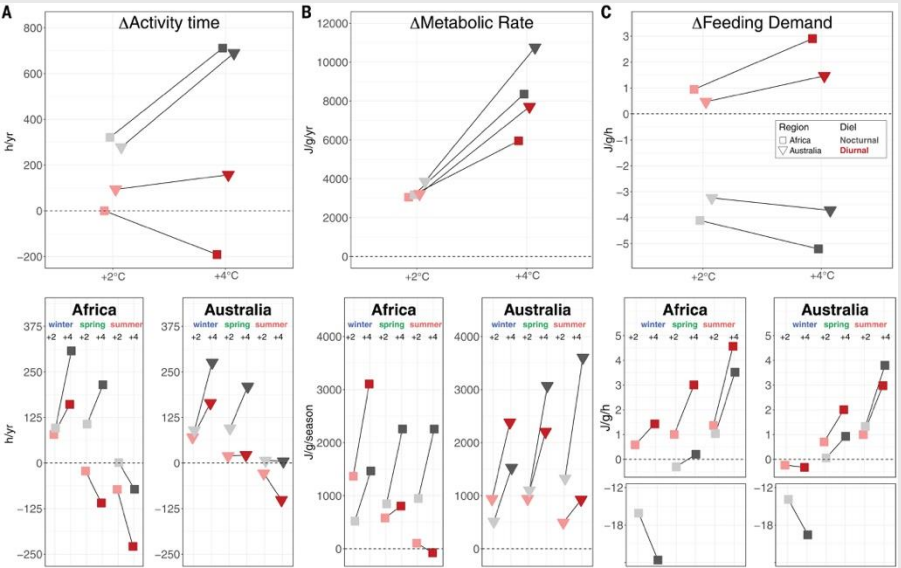


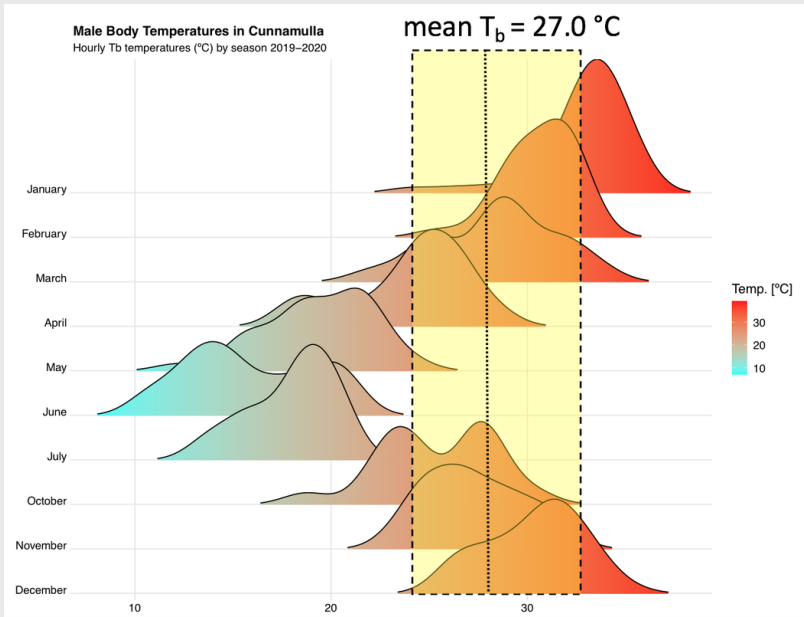


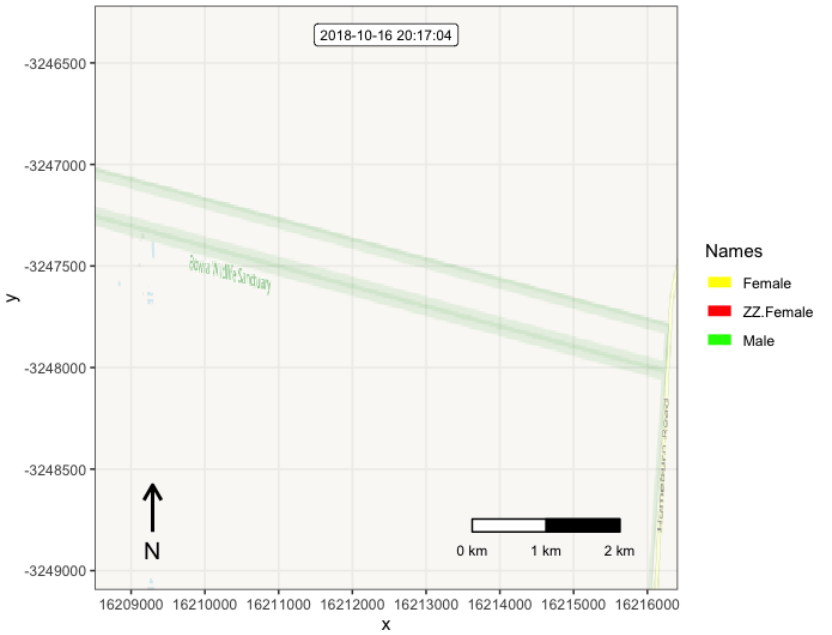
# Kristoffer Wild

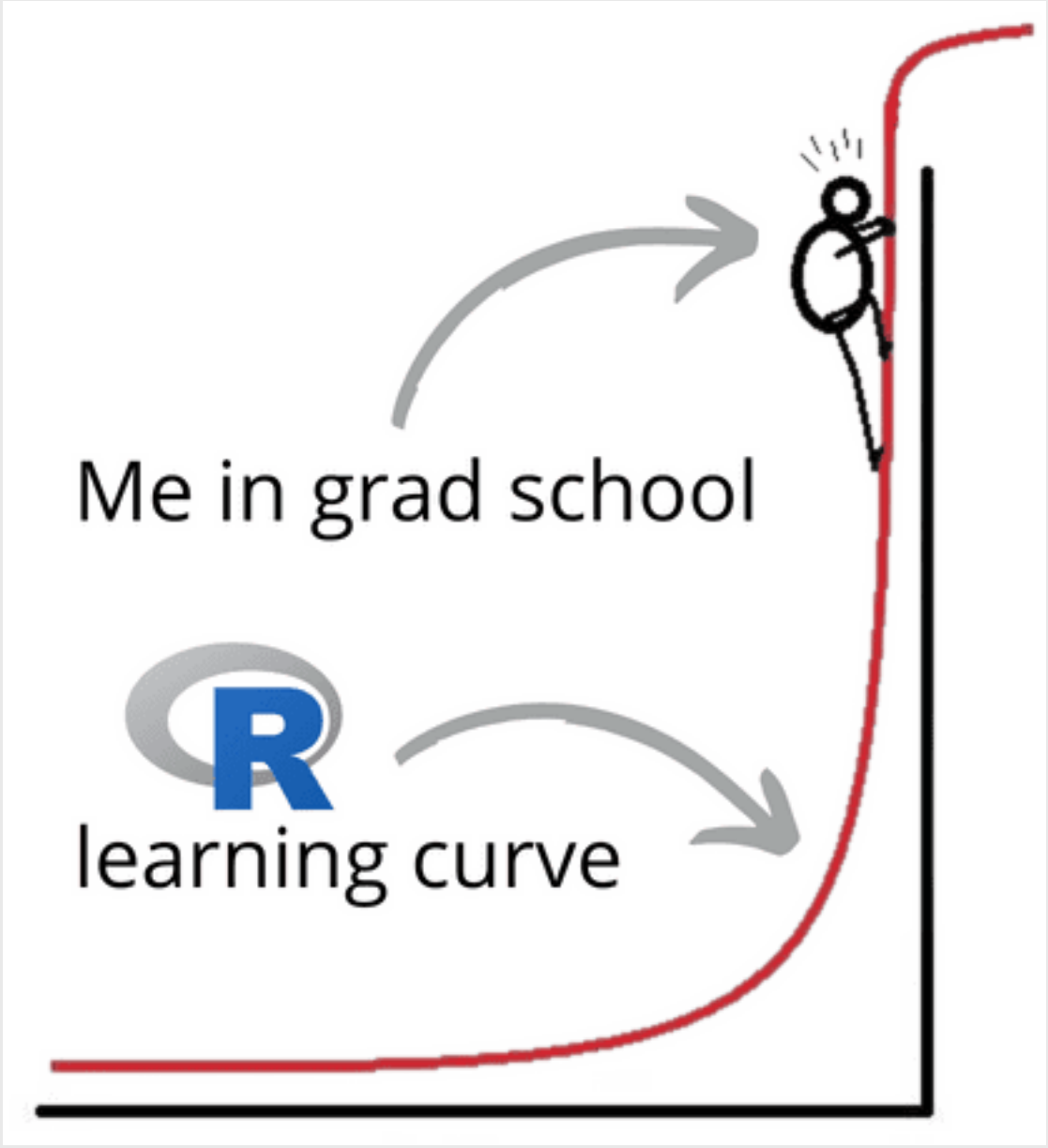
kristofferw@unimelb.edu.au

kwildresearch.com

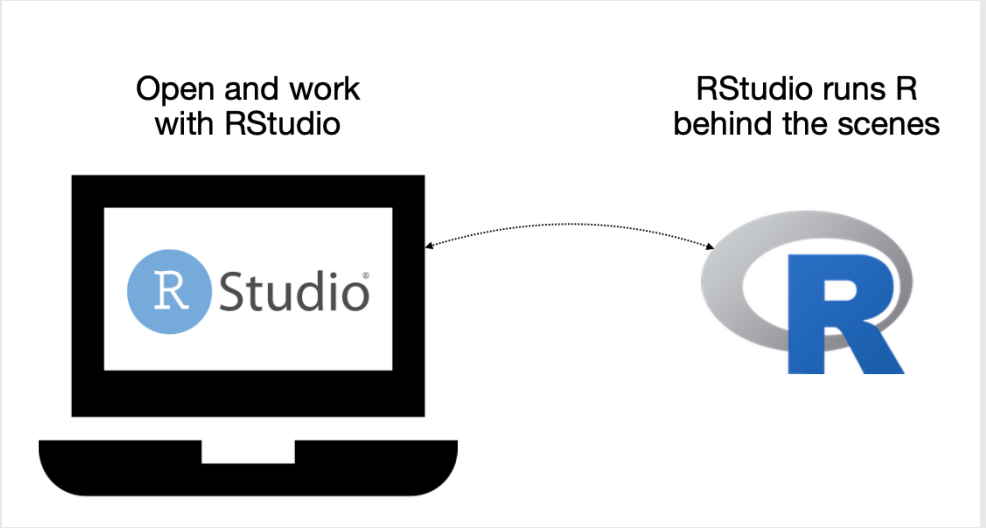



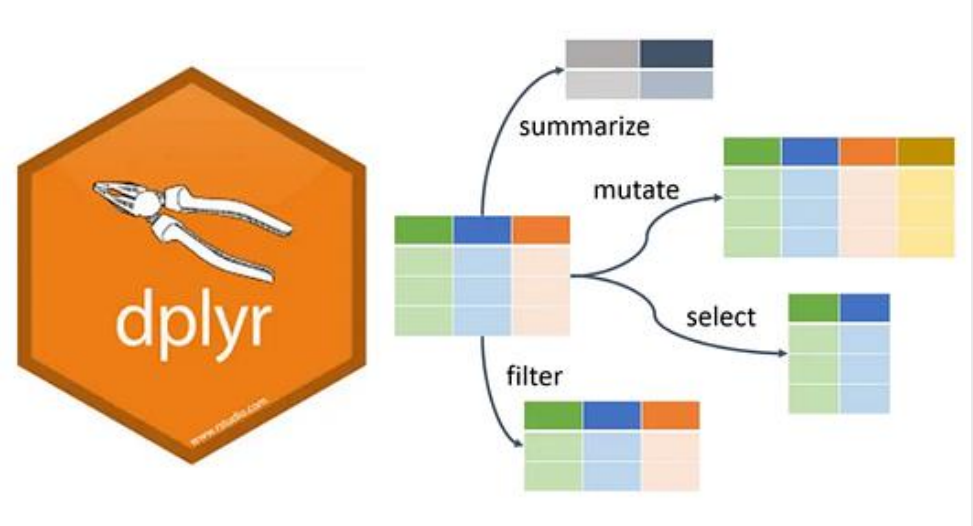

Me in grad school
R
learning curve

# Learning objectives

- Get familiar with R....
  - how to download R and R studio
  - setting up R studio
  - basic R functions
  - data types in R
  - installing a package

- Working with dplyr
  - understanding a data frame
  - working with the common ‘verbs’



# Today's presentation

1. **Introduction to R Programming Language**
2. Structure of R
3. Installation and set-up
4. Data-types in R
5. Working with dplyr

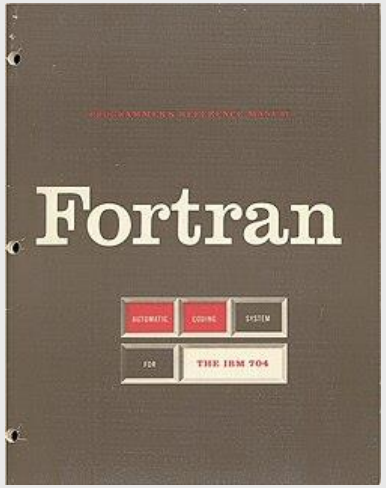
# 1. Introduction to R Programming Language

- R is a computing language that first appeared in the 1990s (c.1993)
- Originally developed at the University of Auckland, New Zealand
- Intended use was for performing statistics and generating graphics
- Use now extends to many other areas:
  - DNA arrays
  - Gene expression arrays
  - RNA-seq analysis
- Predecessor to R was S Programming Language, developed at Bell Laboratories in 1970s
- R has come to be one of the primary computing languages used in bioinformatics

# 1. Introduction to R Programming Language



- R is fundamentally a 'scripting' language, not a high-end programming language
- Therefore it is limited in what it can do
- High-end programming languages include C, C++, JAVA, Fortran
  - these can be used to build complex, mainstream computer applications
- R is mainly for short programs that are interpreted in a batch-like (sequential) manner
- R was created using C and Fortran code



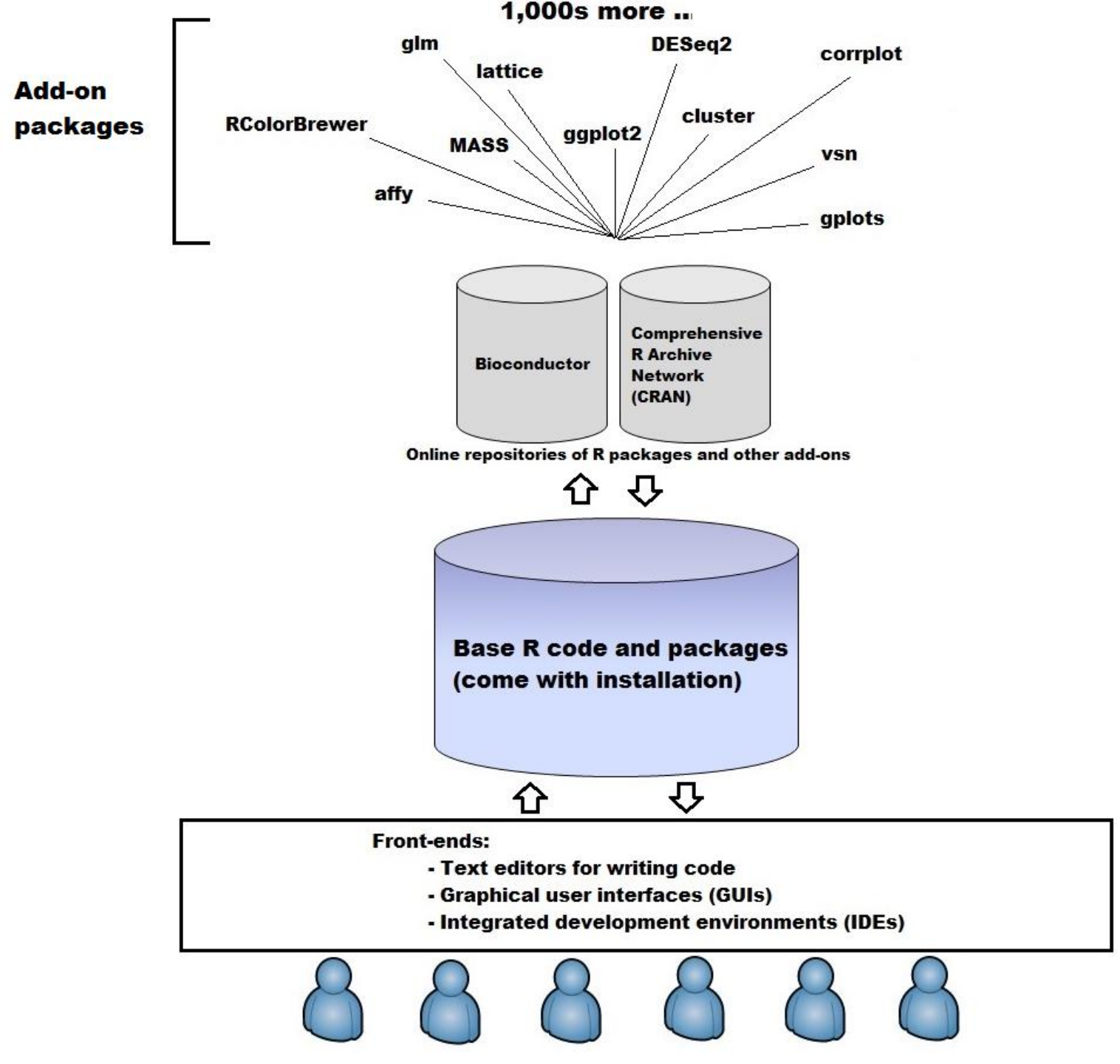

In this lecture, code that you are expected to type is coloured red and 'have #s above it!'

# Today's presentation

1. Introduction to R Programming Language
2. **Structure of R**
3. Installation and set-up
4. Data-types in R
5. Working with dplyr

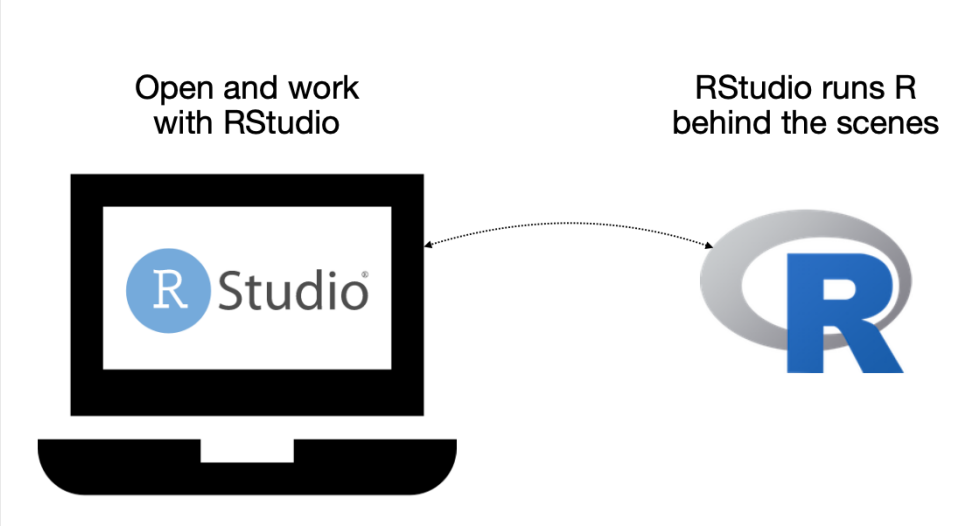
# 2. Structure of R

- The R environment is comprised of 'packages' (1,000s)
- Packages are stored on the World Wide Web in repositories like CRAN (Comprehensive R Archive Network) and Bioconductor
- A package can be thought as a set of functions that perform related tasks
  - dplyr popular data manipulation library
  - stringr For string manipulation.
  - ggplot2 package provides many graphical functions
- When you first install R, many 'core' packages come bundled with the installation; other packages must be manually added


1,000s more ..
Add-on packages
glm
lattice
DESeq2
corrplot
RColorBrewer
MASS
ggplot2
cluster
vsn
affy
gplots
Bioconductor
Comprehensive R Archive Network (CRAN)
Online repositories of R packages and other add-ons
Base R code and packages (come with installation)
Front-ends:
- Text editors for writing code
- Graphical user interfaces (GUIs)
- Integrated development environments (IDEs)

# Today's presentation

1. Introduction to R Programming Language
2. Structure of R
3. **Installation and set-up**
4. Data-types in R
5. Working with dplyr

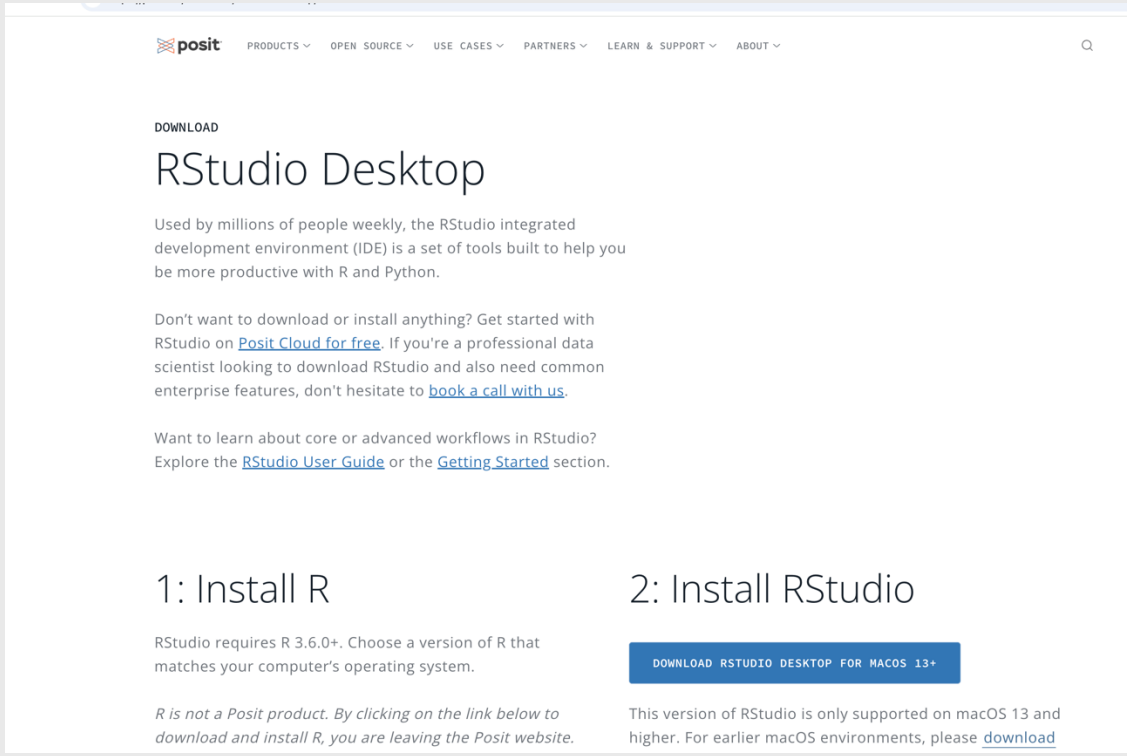


# 3. Installation and set-up

If you have not already, install **R** and **RStudio** on your computer

https://posit.co/download/rstudio-desktop/

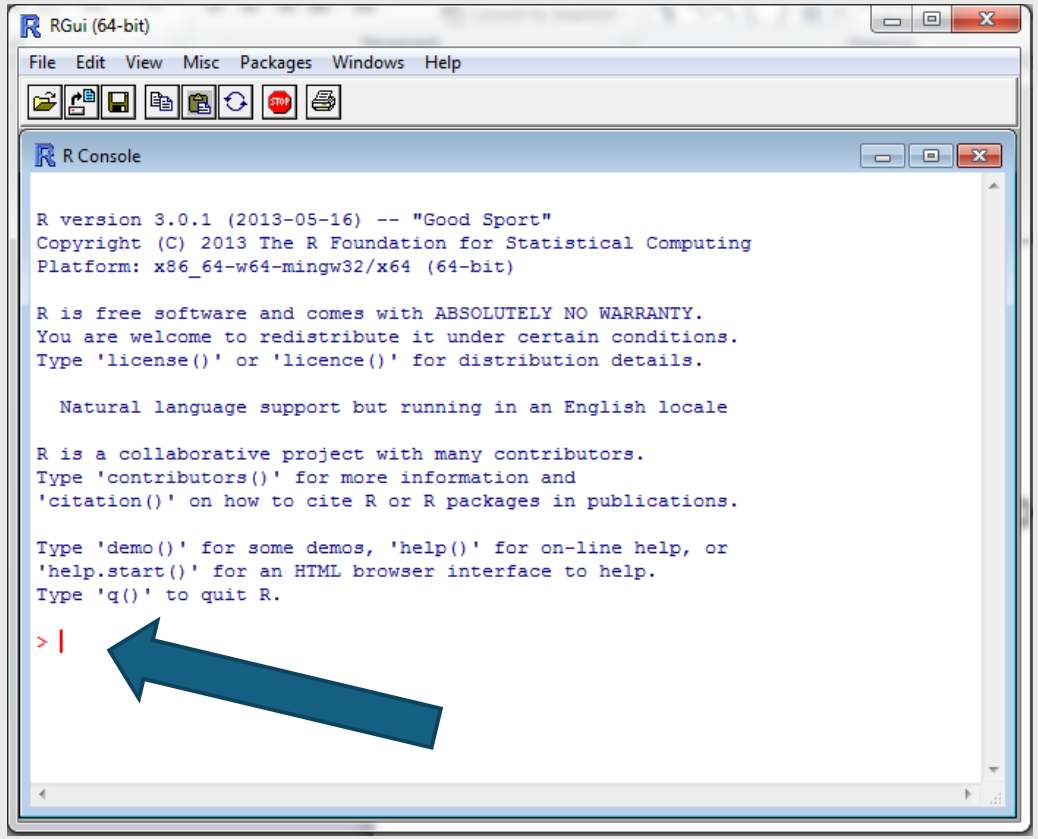
- Choose any mirror, closer ones will be faster



# Load up R

- Find the standalone R program
- Open it
- Enter commands at the > sign, e.g.

```
> 2 + 4
> x <- 7
> x + 19
```




The > is the R command prompt

This font means this is an R command

# What is R doing?

```
> 2+4
[1] 6
> x <- 7
> x + 19
[1] 26
> X + 10
Error: object 'X' not found
```

The **[1]** means the first element of a vector
Even a single number in R is a vector, so “6” is a vector of size 1

**<-** means “assign” in this case “assign the value 7 to the variable x”
Some use = for this purpose but it is frowned upon

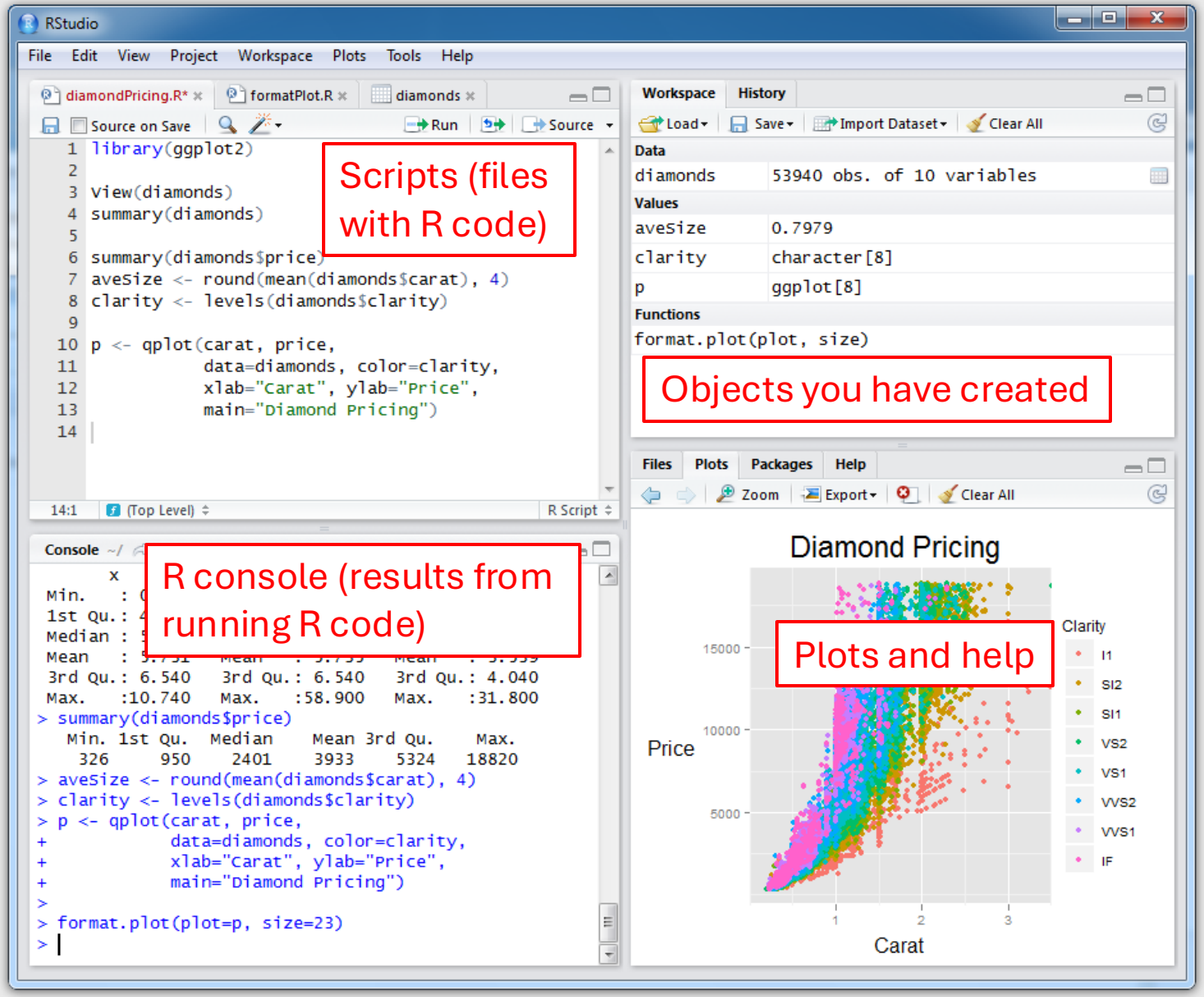
Adding 19 to x gives the expected value of 26

X is not the same as x
R is **case sensitive**: upper case letters are different to lower case letters

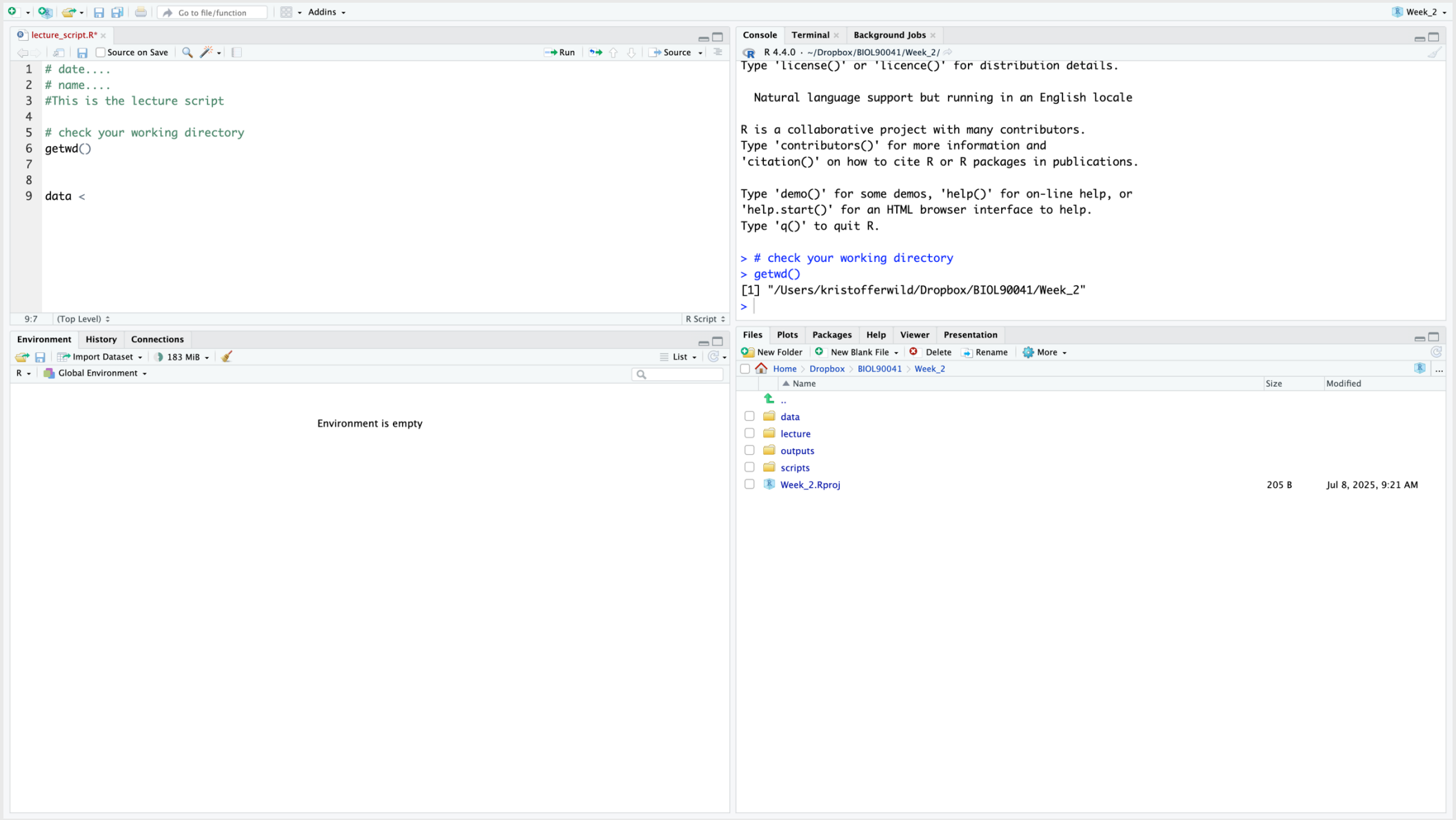
# Rstudio: An Integrated Development Environment

- It is tedious to write R code in the command line
- Old style: create a text file (e.g. Notepad) and copy the code you want to run to the command line
- Much better to use RStudio. Why?
  - Multiple input-output files
  - View variable values, color coding
  - Built-in help
  - Quick running of code
  - Easy file handling
  - Easy package installation
  - Many other reasons

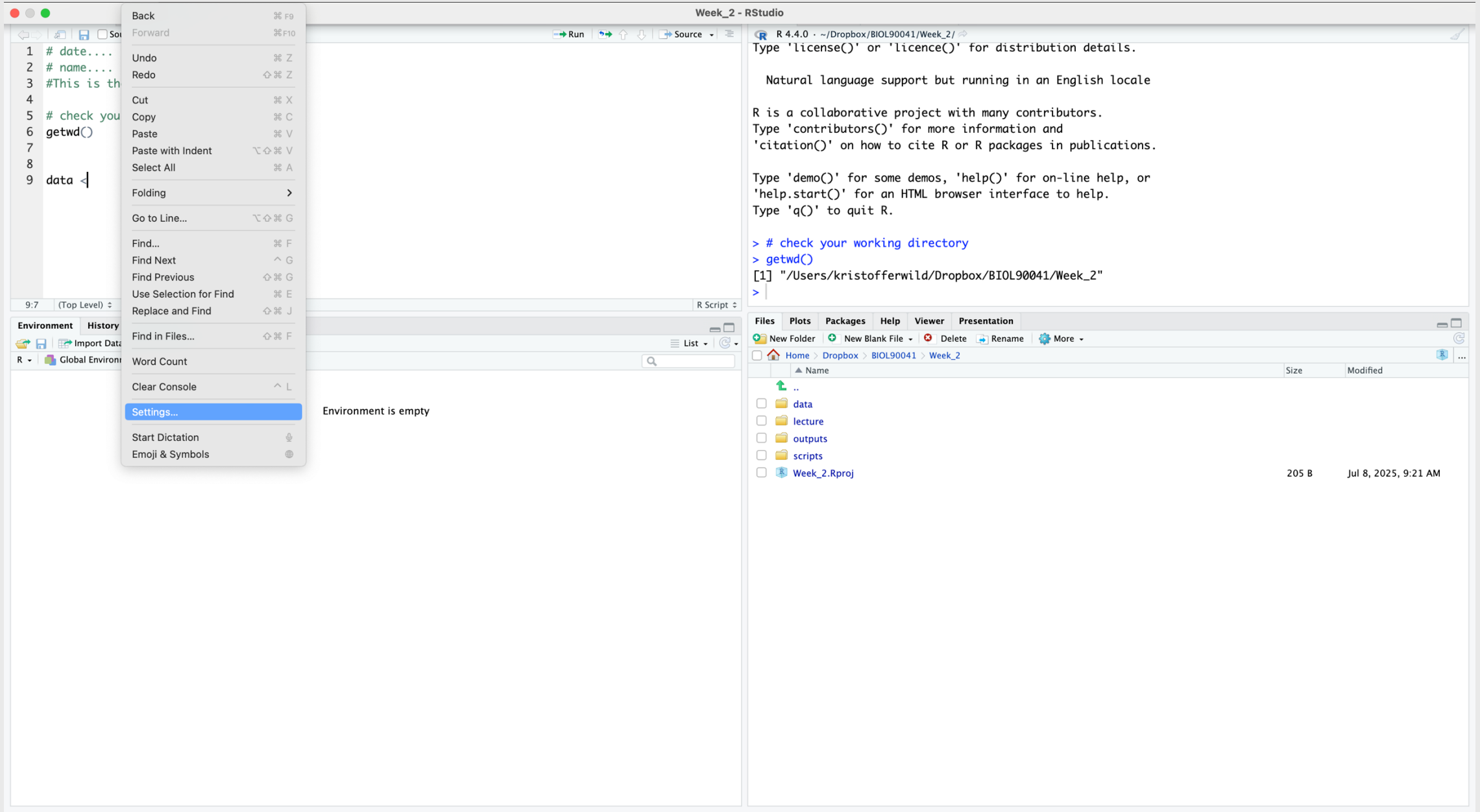
# Rstudio: An example



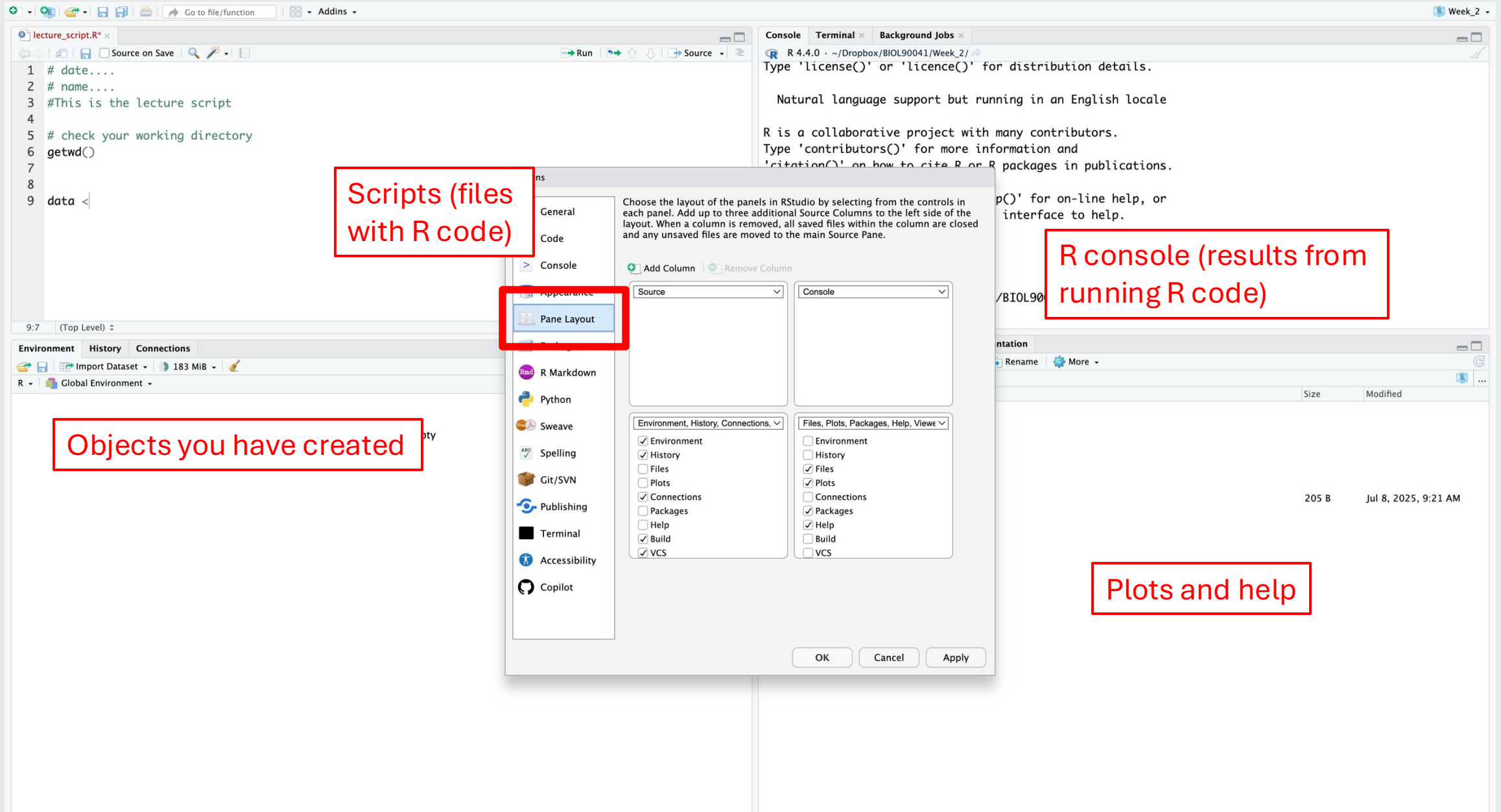
# Rstudio: customise



# Rstudio: customise

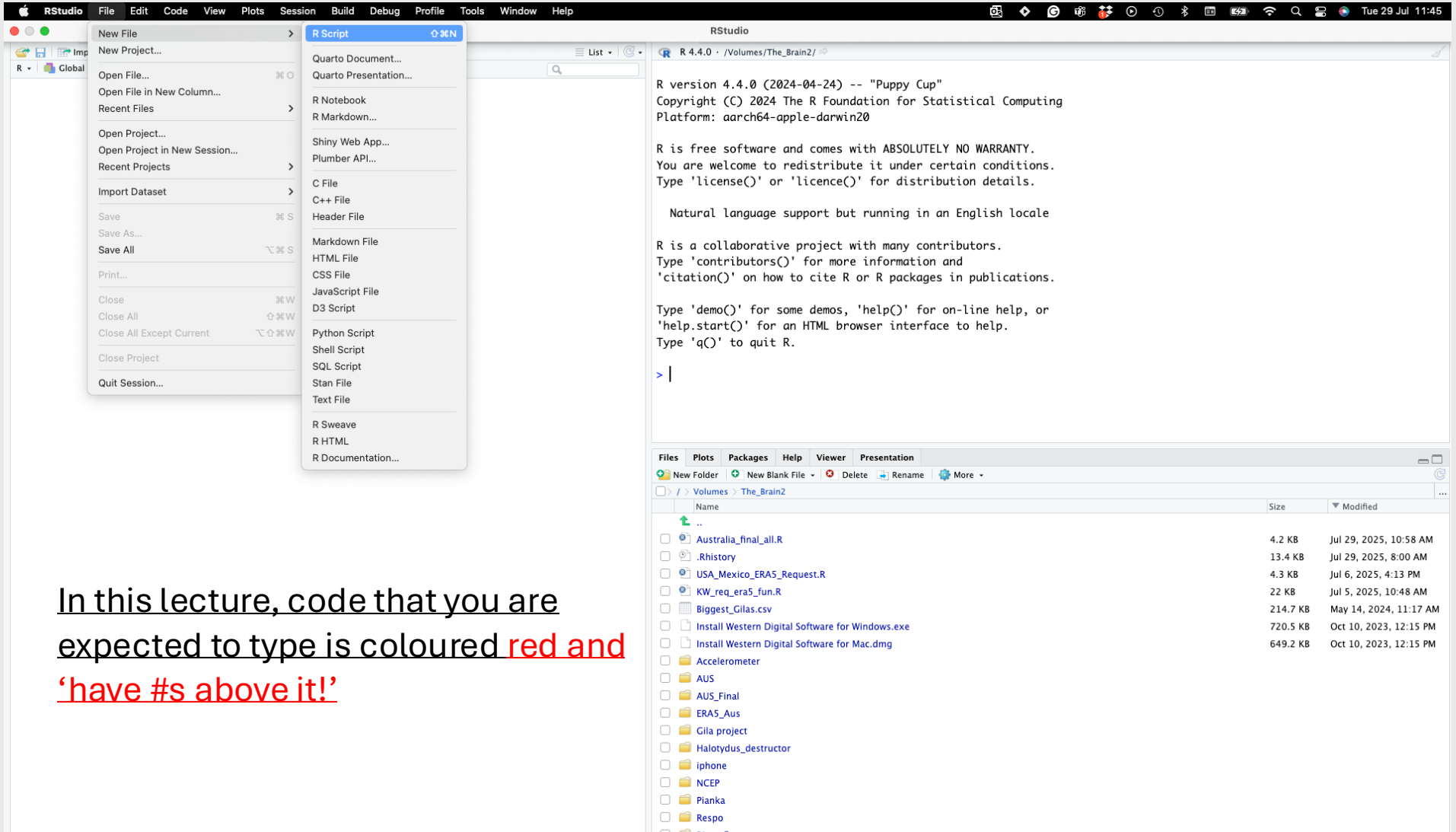


# Rstudio: customise



# 1. Introduction to R Programming Language

Create a script called ‘Lecture2_script.r’

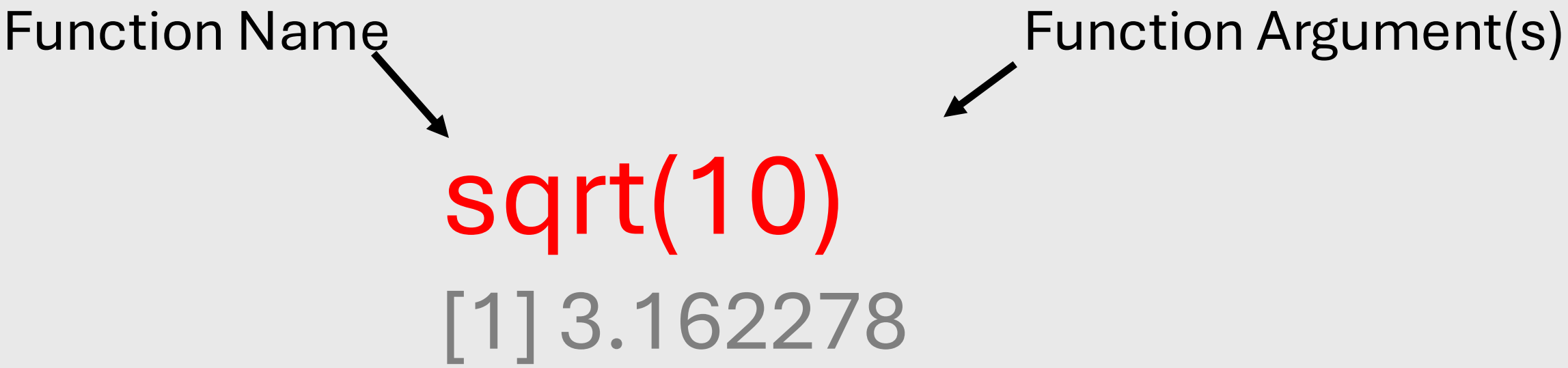



In this lecture, code that you are expected to type is coloured red and ‘have #s above it!’

# R scripts

- A text file (e.g. ***Lecture2_script.r***) that contains all your R code
- It's a complete record of your analyses
- Reproducible: re-running your code is easy for you or someone else
- Easily modified and re-run
- In Rstudio to run code select code and type…
  - '**ctrl+enter'** for pc's (YUCK!)
  - **'command+enter'** for Macs :D
- SAVE YOUR SCRIPTS (unsaved scripts have red title)

# Running a simple function



# Running a simple function

**DID YOU COMMENT AND CODE ???**

```
# function self contained block of code,
# within brackets are arguments or parameters
sqrt(10)
```

```
[1] 3.162278
```

# Help from within R

#Getting help for a function

> ?log()

> help("log")

- Searching across packages

> help.search("logarithm")

- Finding all functions of a particular type

> apropos("log")

[7] "SSlogis" "as.data.frame.logical" "as.logical" "as.logical.factor" "dlogis" "is.logical"

[13] "log" "log10" "log1p" "log2" "logLik" "logb"

[19] "logical" "loglin" "plogis" "print.logLik" "qlogis" "rlogis"

# ?log

R: Logarithms and Exponentials ▾ Find in Topic

log {base} R Documentation

## Logarithms and Exponentials

### Description

What the function does in general terms

`log` computes logarithms, by default natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.

`log1p(x)` computes *log(1+x)* accurately also for |*x*| << *1* (and less accurately when *x is approximately -1*).

`exp` computes the exponential function.

`expm1(x)` computes *exp(x) - 1* accurately also for |*x*| << *1*.

### Usage

How to use the function

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
expm1(x)
```

### Arguments

What does the function need

| | |
|---|---|
| `x` | a numeric or complex vector. |
| `base` | a positive or complex number: the base with respect to which logarithms are computed. Defaults to *e*=`exp(1)`. |

### Details

All except `logb` are generic functions: methods can be defined for them individually or via the `Math` group generic.

`log10` and `log2` are only convenience wrappers, but logs to bases 10 and 2 (whether computed *via* `log` or the wrappers) will be computed more efficiently and accurately where supported by the OS. Methods can be set for them individually (and otherwise methods for `log` will be used).

`logb` is a wrapper for `log` for compatibility with S. If (S3 or S4) methods are set for `log` they will be dispatched. Do not set S4 methods on `logb` itself.

All except `log` are primitive functions.

R: Logarithms and Exponentials ▾ Find in Topic

### Value

What does the function return

A vector of the same length as `x` containing the transformed values. `log(0)` gives `-Inf`, and `log(x)` for negative values of `x` is `NaN`. `exp(-Inf)` is `0`.

For complex inputs to the log functions, the value is a complex number with imaginary part in the range *[-pi, pi]*: which end of the range is used might be platform-specific.

### S4 methods

`exp`, `expm1`, `log`, `log10`, `log2` and `log1p` are S4 generic and are members of the `Math` group generic.

Note that this means that the S4 generic for `log` has a signature with only one argument, `x`, but that `base` can be passed to methods (but will not be used for method selection). On the other hand, if you only set a method for the `Math` group generic then `base` argument of `log` will be ignored for your class.

### Source

`log1p` and `expm1` may be taken from the operating system, but if not available there are based on the Fortran subroutine `dlnrel` by W. Fullerton of Los Alamos Scientific Laboratory (see http://www.netlib.org/slatec/fnlib/dlnrel.f and (for small `x`) a single Newton step for the solution of `log1p(y) = x` respectively.

### References


Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for `log`, `log10` and `exp`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (for `logb`.)


### See Also

Discover other related functions

`Trig`, `sqrt`, `Arithmetic`.

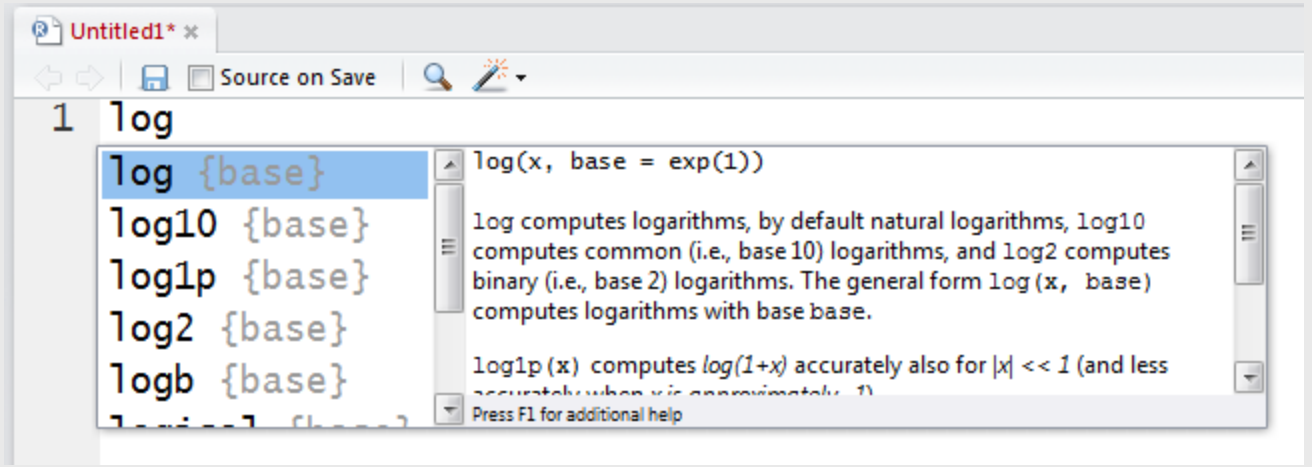
### Examples

Sample code showing how it works

```
log(exp(3))
log10(1e7) # = 7

x <- 10^-(1+2*1:9)
cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))
```

# RStudio quick help

- Start typing in the Scripts window (top-left)
- Press <tab> and a list of available functions starting with those letters appears, plus help




- Try typing log( and then pressing <tab>

# STOP AND CODE

Say you have this object below

cool_vec <- ‘kris is cool’

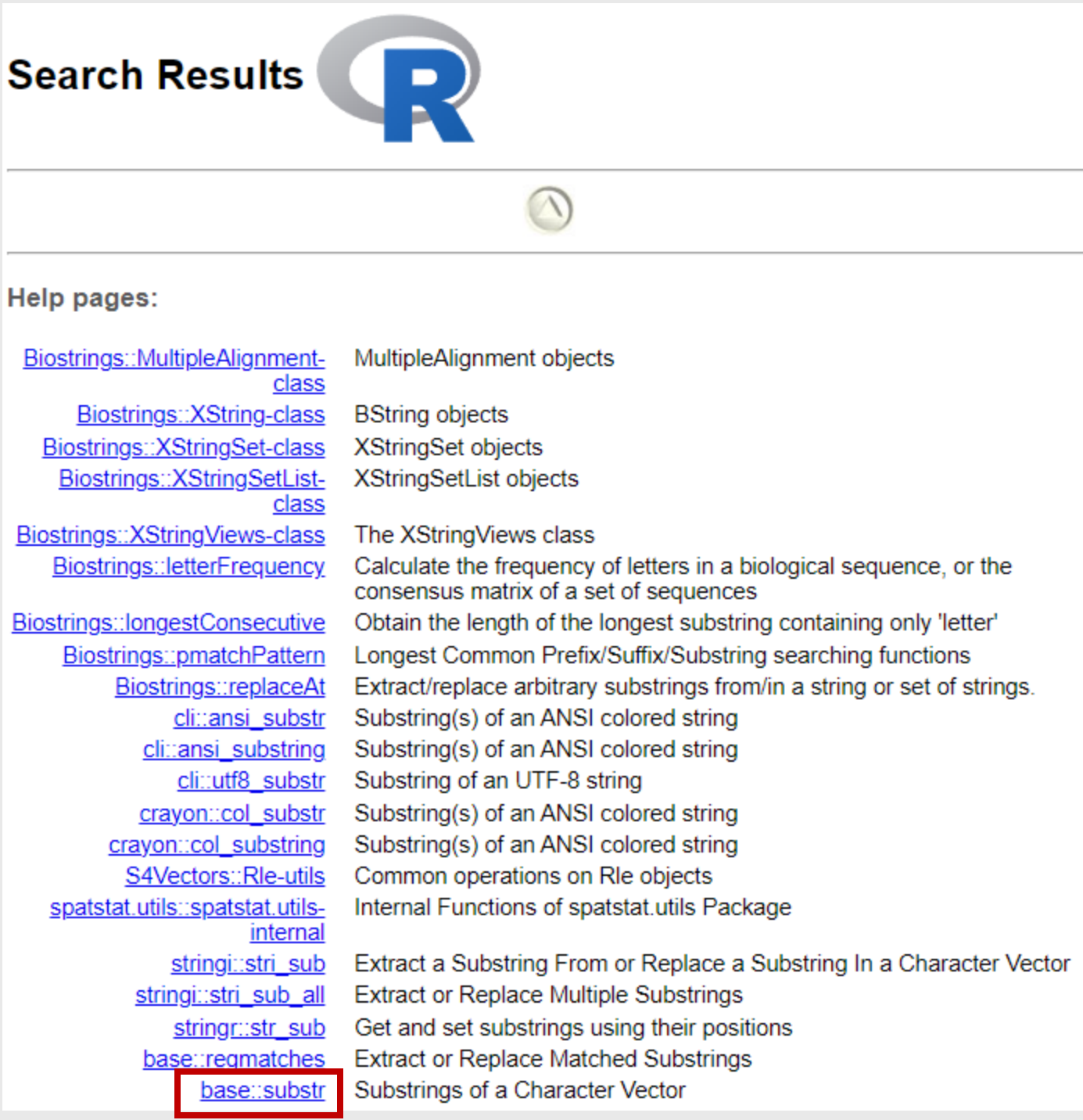
Now not knowing the **rep()** function, figure out how  repeat that object 3 times using that function

# Searching Help

```
??substring
```

Package → `base`

Function → `::substr`

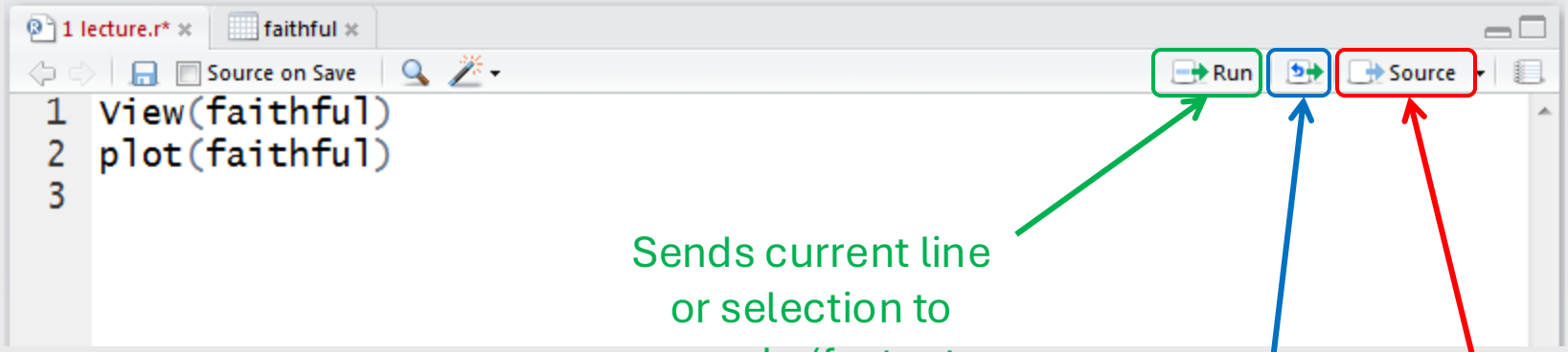


## Search Results

**Help pages:**

| | |
|---|---|
| Biostrings::MultipleAlignment-class | MultipleAlignment objects |
| Biostrings::XString-class | BString objects |
| Biostrings::XStringSet-class | XStringSet objects |
| Biostrings::XStringSetList-class | XStringSetList objects |
| Biostrings::XStringViews-class | The XStringViews class |
| Biostrings::letterFrequency | Calculate the frequency of letters in a biological sequence, or the consensus matrix of a set of sequences |
| Biostrings::longestConsecutive | Obtain the length of the longest substring containing only 'letter' |
| Biostrings::pmatchPattern | Longest Common Prefix/Suffix/Substring searching functions |
| Biostrings::replaceAt | Extract/replace arbitrary substrings from/in a string or set of strings. |
| cli::ansi_substr | Substring(s) of an ANSI colored string |
| cli::ansi_substring | Substring(s) of an ANSI colored string |
| cli::utf8_substr | Substring of an UTF-8 string |
| crayon::col_substr | Substring(s) of an ANSI colored string |
| crayon::col_substring | Substring(s) of an ANSI colored string |
| S4Vectors::Rle-utils | Common operations on Rle objects |
| spatstat.utils::spatstat.utils-internal | Internal Functions of spatstat.utils Package |
| stringi::stri_sub | Extract a Substring From or Replace a Substring In a Character Vector |
| stringi::stri_sub_all | Extract or Replace Multiple Substrings |
| stringr::str_sub | Get and set substrings using their positions |
| base::regmatches | Extract or Replace Matched Substrings |
| base::substr | Substrings of a Character Vector |

# RStudio tips




Sends current line or selection to console (faster to type ctrl+enter)

Re-send the lines of code you last ran (useful after edits)

Sends entire file to R console

# Commenting your code (do it)

- Use “comments” to document the intention of your code
- Anything on a line after # is ignored by R

```
# Old Faithful Geyser, Yellowstone NP

plot(faithful)
```

RStudio: different color for comments

- Rules of thumb
  - Document the purpose of the code not how it works
  - Use good variable names
  - Assume you will remember nothing about the code when you look at it later (next week, year, decade)

# Some simple R commands

```
> 2+2
[1] 4
> 3^2
[1] 9
> sqrt(25)
[1] 5
> 2*(1+1)
[1] 4
> 2*1+1
[1] 3
```

Result of the command

Order of precedence

```
> exp(1)
[1] 2.718282
> log(2.718282)
[1] 1
> log(10, base=10)
[1] 1
> log(10
+ , base = 10
[1] 1
>rep()
```

Optional argument

Incomplete command

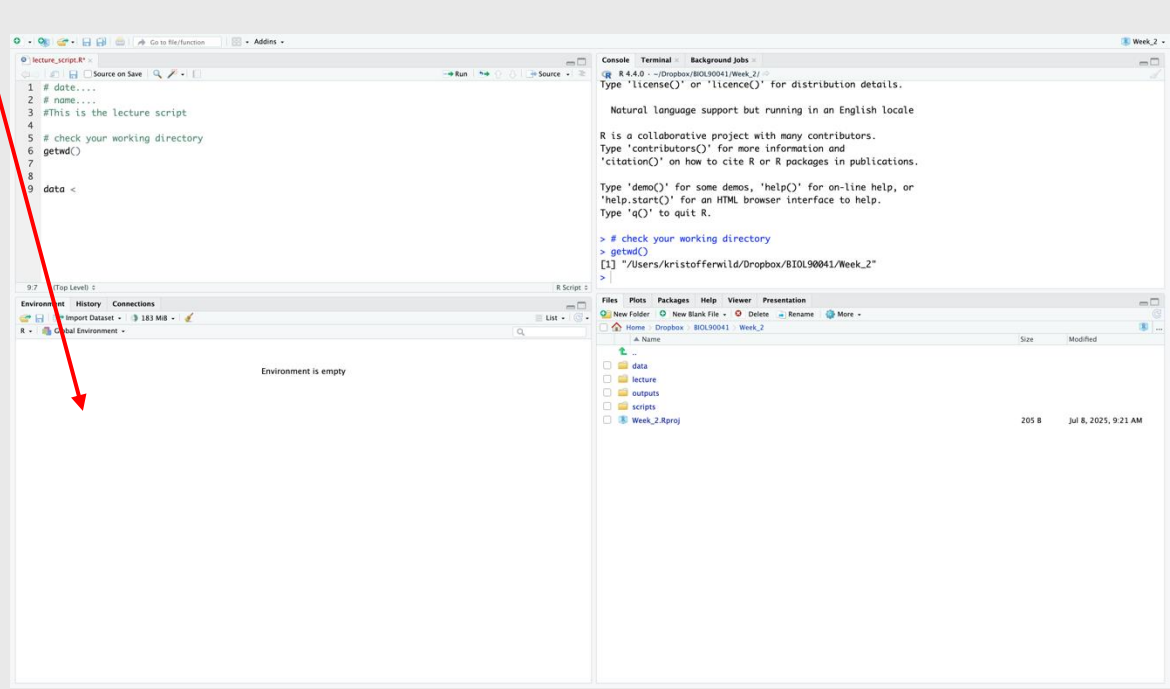
# Frequently used operators

| | |
|---|---|
| <- | Assign |
| + | Sum |
| - | Difference |
| * | Multiplication |
| / | Division |
| ^ | Exponent |
| %% | Mod |
| %*% | Dot product |
| %/% | Integer division |
| %in% | Subset |

| | |
|---|---|
| \| | Or |
| & | And |
| < | Less |
| > | Greater |
| <= | Less or = |
| >= | Greater or = |
| ! | Not |
| != | Not equal |
| == | Is equal |

# R workspaces

- When you close your R session, you can save data and analyses in an R workspace
- This saves everything run in your R console
- Generally not recommended
  - Exception: working with an enormous dataset
- Better to start with a clean, empty workspace so that past analyses don't interfere with current analyses
- rm(list = ls()) clears out your workspace
- Summary: save your R script, don't save your workspace

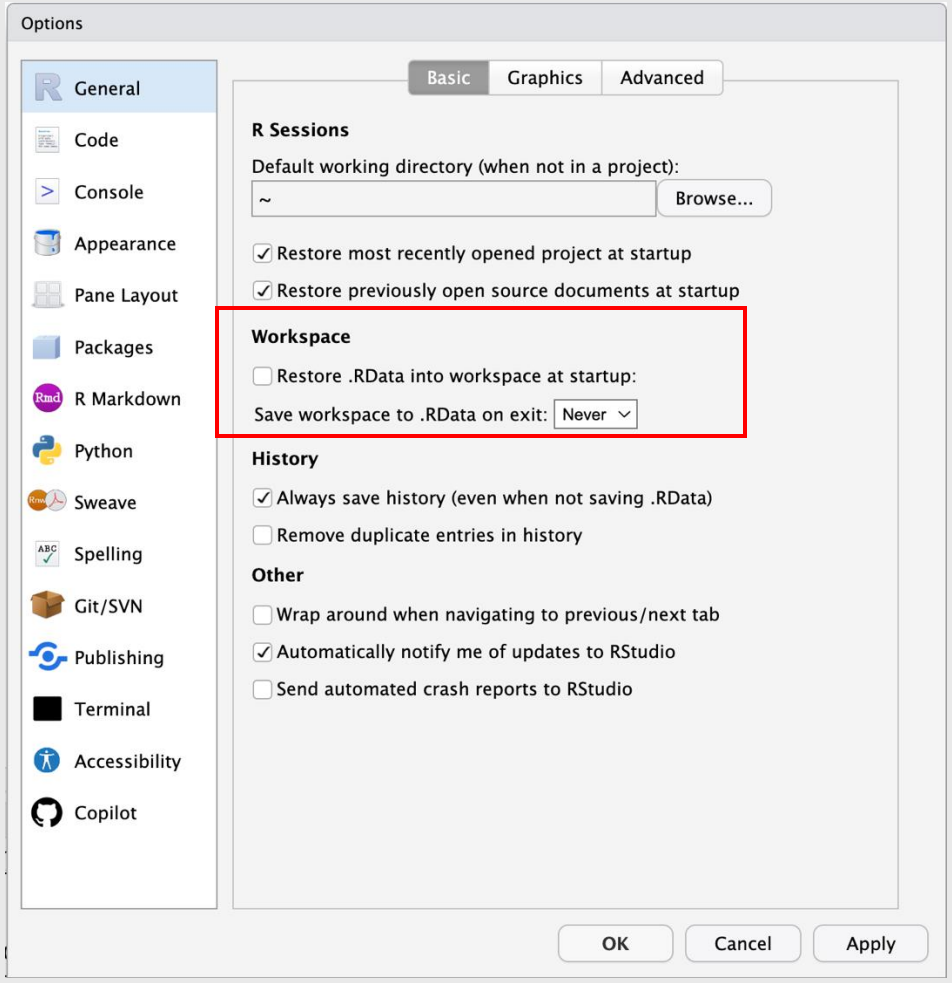


# R workspaces

Turn off default saving/restoration of workspace

Edit → Settings…

Turn off "*Restore .Rdata into workspace at startup*"



# STOP AND CODE

- Use R to do the following. Using your '***Lecture2_script.r***' answer the following. Remember to save & **comment '#' your work**:

1) $1 + 2(3 + 4)$

2) $\ln(4^3+3^{2+1})$

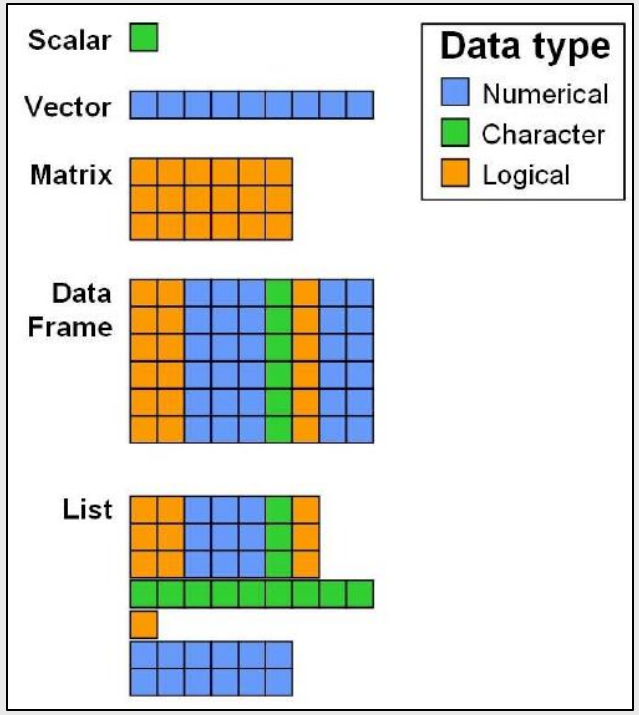
3) $\sqrt{(4+3)(2+1)}$

4) $\left(\frac{1+2}{3+4}\right)^2$

Try running these!

# Objects

- Every programming outcome in R can be stored as an **object**
  - Numbers
  - Characters (i.e. text or strings)
  - Tables
  - Vectors and matrices
  - Plots
  - Statistical output
  - Functions
- Good names for objects are critical
- Objects in R are **global**



# Assigning values

# Assign the result of log(2.5) to a new object called “answer”

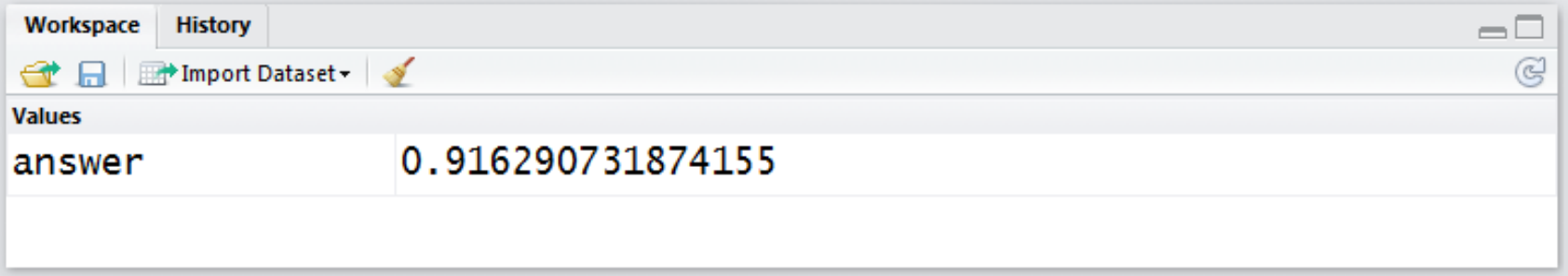
```
answer <- log(2.5)
```

# = can be used instead of <- but is frowned upon;
# YUCK!!!

```
answer = log(2.5)
```

# optional argument

```
answer <- log(2.5, base=10)
```




When you run this command, an object “answer” is created in the workspace that is assigned the value of 0.91629… In RStudio, the workspace window lists all the objects in the current workspace

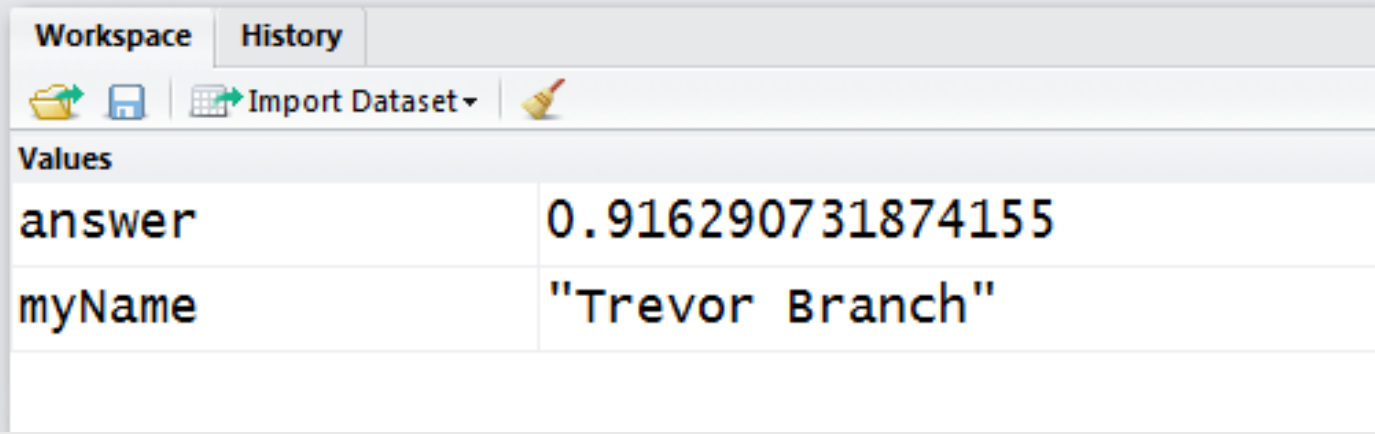
# Assigning values

# Characters can also be assigned to objects

```
myName <- "Trevor"
```

# Usually we use double quotation marks, but
# single quotes are treated the same

```
myName <- 'Trevor'
```




# Surrounding a command in ( ) will display
# the assigned value in the R console; Can include spaces

```
> (myName <- "Trevor Branch")
[1] "Trevor Branch"
```

# Viewing objects

- RStudio: just look at the top-right Workspace tab
- Alternatively (and more generally):

```
# Very general command, works on everything
> print(answer)
[1] 0.9162907

> answer
[1] 0.9162907

# Manipulate the value contained within the object
> answer * 10
[1] 9.162907
```

# Removing objects

# To find a list of all objects in the workspace

```
> ls()
[1] "answer" "myName"
```

# To remove an object

```
> rm(answer)
> ls()
[1] "myName"
```

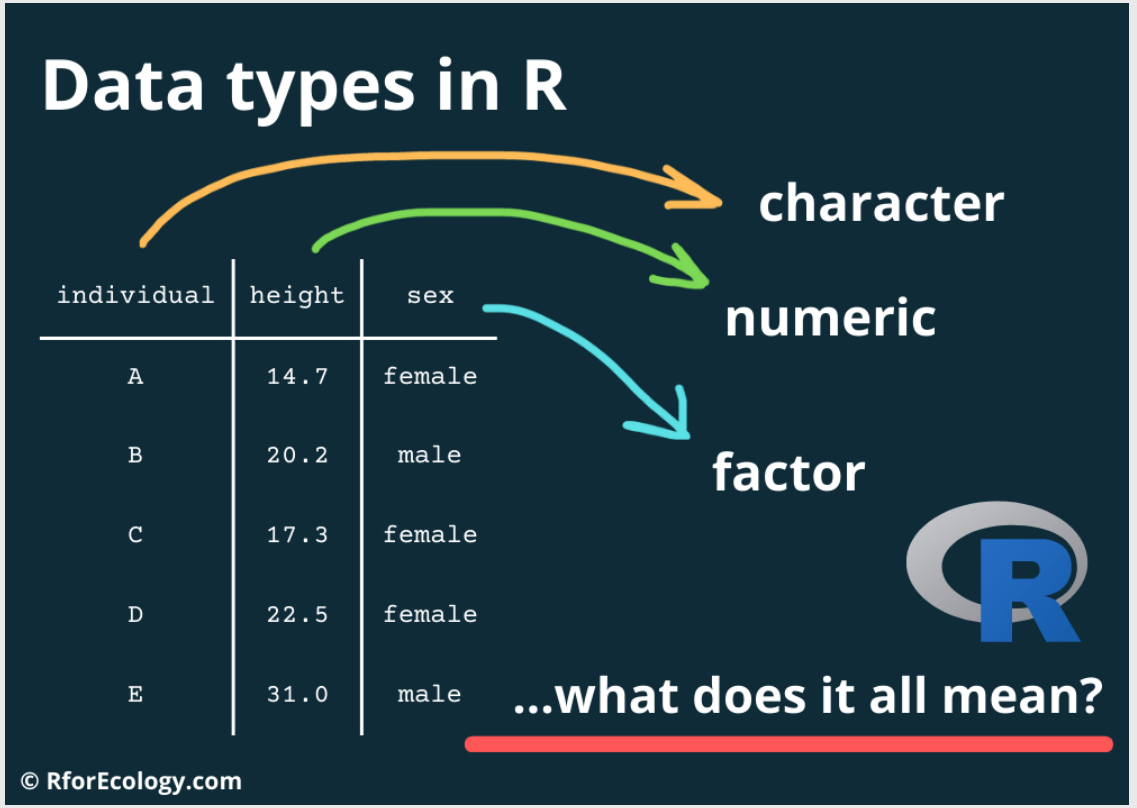
- To remove **all** objects

```
> rm(list = ls())
> ls()
character(0)
```

Nothing left in workspace

# Today's presentation

1. Introduction to R Programming Language
2. Structure of R
3. Installation and set-up
4. **Data-types in R**
5. Working with dplyr



# Data types

- Data types describe how objects are stored in computer memory
- In R, you do **not** need to specify the data type
- Common data types (also known as **mode**) include
  - Numeric (integer, floating point numbers or doubles)
  - Logical (Boolean, true or false)
  - Characters (text or string data)
- The object type is not always obvious in R, and knowing what it is can be important

# Finding data types??

```
> answer <- log(2.718282)
> answer
[1] 1
# Is it numeric or text?
> mode(answer)
[1] "numeric"
# Part of a family of is. functions
> is.numeric(answer)
[1] TRUE
```

Specifically, what type of object is it?

```
> typeof(answer)
[1] "double”
```

The as. functions **coerce** objects from one type to another

```
> answer <- as.integer(answer)
> typeof(answer)
[1] "integer"
```

# Everything is a vector

- Vectors are the most basic unit of storage in R

- Vectors are ordered sets of values of the same type
  - Numeric
  - Character (text)
  - Logical (TRUE or FALSE)
  - Date etc…

```
10 -> x
```

x is a vector of length 1 with 10 as its first value

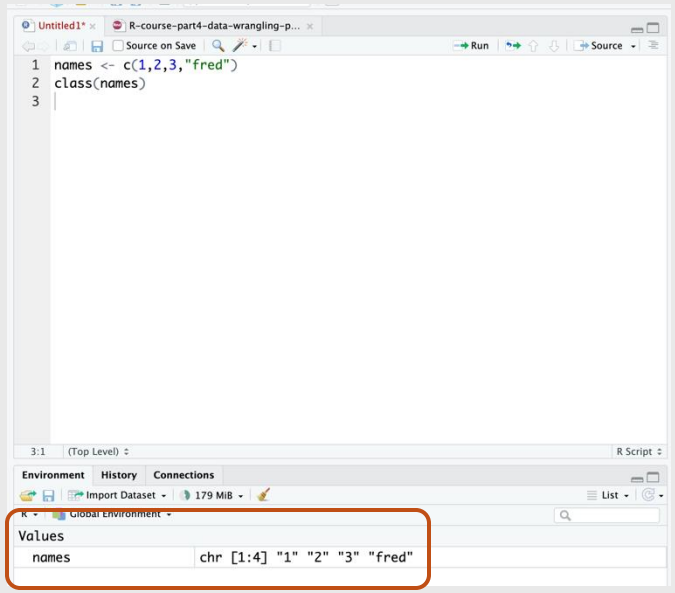
# Creating vectors manually

#Use the c (combine) function

```
c(1,2,4,6,3) -> simple.vector

c("simon","laura","sarah","jo","louise") -> some.names
```

#Data must be of the same type

```
names <- c(1,2,3,"fred")
[1] "1"    "2"    "3"    "fred"
```




**Question: What was the class of the object 'names'?**

# Catching up on the basics

- If this was a lot just take a look at this video for some extra practice:

# Data Frames and Lists

- Multivariate data consists of multiple data vectors considered as a whole.

- We’re free to work with our data as separate variables, but it can behoove us to combine related vectors into a single object such as a data frame, which is used to store a rectangular grid of data.
  - Usually, rows correspond to experimental units/subjects, and columns correspond to variables/measurements.
  - All column lengths are the same.

- A list is a more general storage object which is comprised of a collection of components
  - Size of the objects can differ.
  - Anything goes in terms of structure!

# Examples of Both

- **Data frame**: Any one of the various data sets we will be using
- **List**: Typically the output of a function. E.g., consider the object
  - Dhf = density(iris$Sepal.length)
  - We can check that Dhf is a list via summary(Dhf), which tells us each component and gives its data type.

# Creating a Data Frame or List

#Data frames are created with data.frame(). E.g.,

- x<-1:2
- y<-letters[1:2]
- z<-1:3
- data.frame(x,y)
- data.frame(x,y,z) #error due to unequal lengths

#Lists are created with list(). E.g.,

- list(x,y,z) #unnamed components
- list(x.name=x,y.name=y,z.name=z) #named components

# Accessing Values in a Data Frame

# Can access values with either the name of a column or with row column notation

- DF <- data.frame(x,y)
- DF$x #access x column of DF
- DF$[,'x'] #access column named x
- DF$x[1] #access first entry in column x of DF
- DF[1,2] # access entry in first row, second column

# Accessing values in a list

#If list components are unnamed, use double-bracket notation [[]].

- lst = list(x,y,z)
- lst[[1]] #access first component x

#If components are named, access using $comp.name.

- lst.named = list(x.name=x, y.name=y, z.name=z)
- lst.named$x.name
- names(lst.named)
- lst.named[['x.name']]

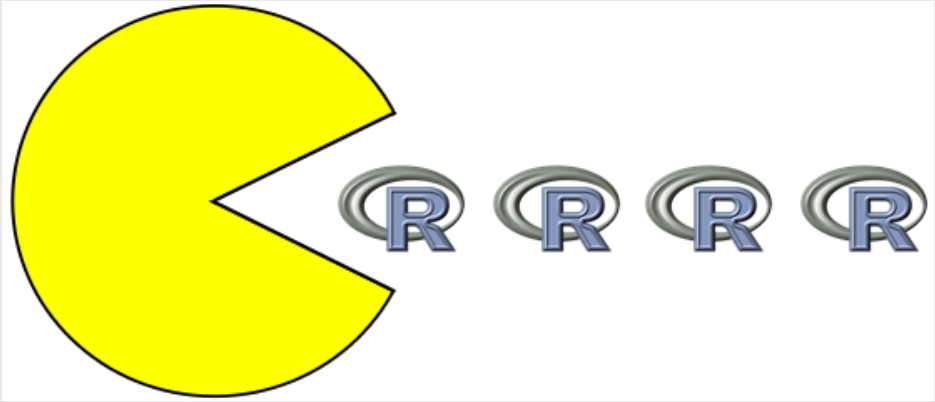
- For both data frames and lists, can edit values or components by accessing them and reassigning them, or using edit(obj).

# Today's presentation

1. Introduction to R Programming Language
2. Structure of R
3. Installation and set-up
4. Data-types in R
5. **Working with dplyr**

# Data manipulation with **dplyr**

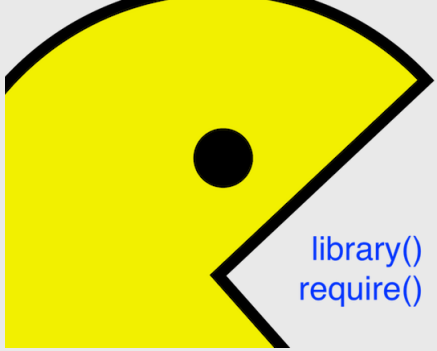
# But first…. We use pacman to load and upload packages



**What is pacman???**

- Combines and simplifies base R package functions with p_xxx commands
- Speeds up workflows and reduces code when added to .Rprofile
- p_load() loads and installs missing packages in one step
- Most functions don’t require quotes around package names
- Ideal for scripts, teaching, and sharing reproducible code

# Install pacman




**#Install pacman if not already installed**

- install.packages("pacman")

#Now use **p_load()** to load dplyr and nycflights13 (installs if missing)

- pacman::p_load(dplyr, nycflights13)

# you will now use the function p_load() and any package at the start of all of your scripts!


**Quick Reference Table**

| Pacman Function | Base Equivalent | Description |
|---|---|---|
| `p_load` | `install.packages` + `library` | Loads and Install Packages |
| `p_install` | `install.packages` | Install Packages from CRAN |
| `p_load_gh` | NONE | Loads and Install GitHub Packages |
| `p_install_gh` | NONE | Install Packages from GitHub |
| `p_install_version` | `install.packages` & `packageVersion` | Install Minimum Version of Packages |
| `p_temp` | NONE | Install a Package Temporarily |
| `p_unload` | `detach` | Unloads Packages from the Search Path |
| `p_update` | `update.packages` | Update Out-of-Date Packages |

# dplyr and Tibbles


- dplyr Contstrains options, helps one think about data manipulation challenges.
- Provides simple “verbs”, functions that correspond to common data manipulation tasks
- Works efficiently and quickly
- Utilizes objects known as **tibbles**, which can be thought of as modernized data frames
- E.g., flights (nycflights13) is saved as a tibble object
- To convert a data frame to a tibble, use as_tibble()

# Single Table Verbs


- dplyr aims to provide a function for each basic task of data manipulation.
- As in base R, all the output of all of these functions can be stored as an object in the workspace.

| | |
|---|---|
| filter() | Select cases based on their values |
| arrange() | Reorder based on values of a variable |
| select() and rename() | Select variables based on their names. |
| mutate() and transmute() | Add new variables that are functions of existing variables. |
| summarise() | Condense multiple values to a single value |
| sample_n() and sample_frac() | Take random samples. |

# Data: nycflights13

- We'll explore all this through the example of the flights (nycflights13) data set, which contains info on all 336,776 flights that departed from NYC in 2013.
- Install the package.
  - install.packages(nycflights13)
  - library(nycflights13)
  - flights #inspect data set

**Stop and check**

- Is flights a data frame or a list?
- How many columns are there?
- How many rows?
- how many columns are numeric?

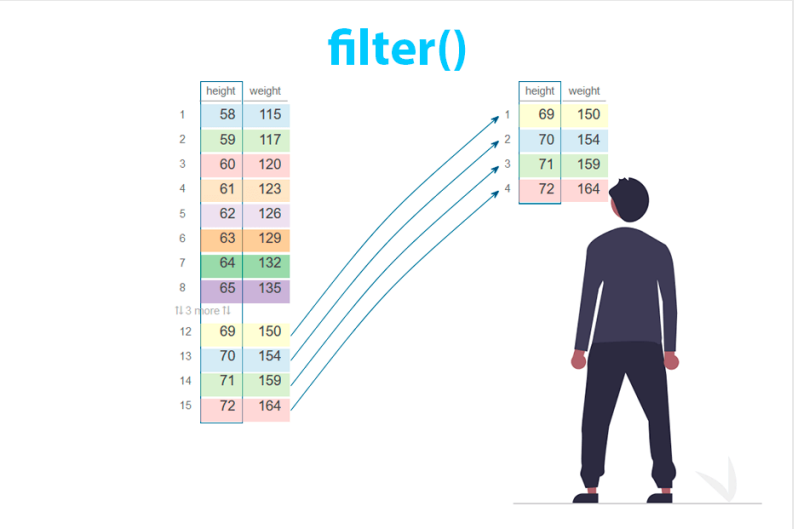
# filter()

- filter() allows us to **select a subset of a tibble**. The first argument is the tibble object, and the subsequent arguments are logical expressions for the data, separated by commas.

- The function returns all data for which all logical statements are TRUE.

# E.g., select all flights on January 1st:
  - filter(flights, month==1,day==1)

- Roughly equivalent to R-base code
  - flights[flights$month==1 & flights$day==1]… Yuck




- When using dplyr functions, do not need to use attach() or $ notation.

# arrange()

- arrange() works similarly to filter(), except that it **reorders rows according to the ordering of one or more particular variables** instead of subsetting them.

- First argument is the tibble, the second argument is the reordering variable, and subsequent arguments are variables that break ties of the variable immediately preceding them.

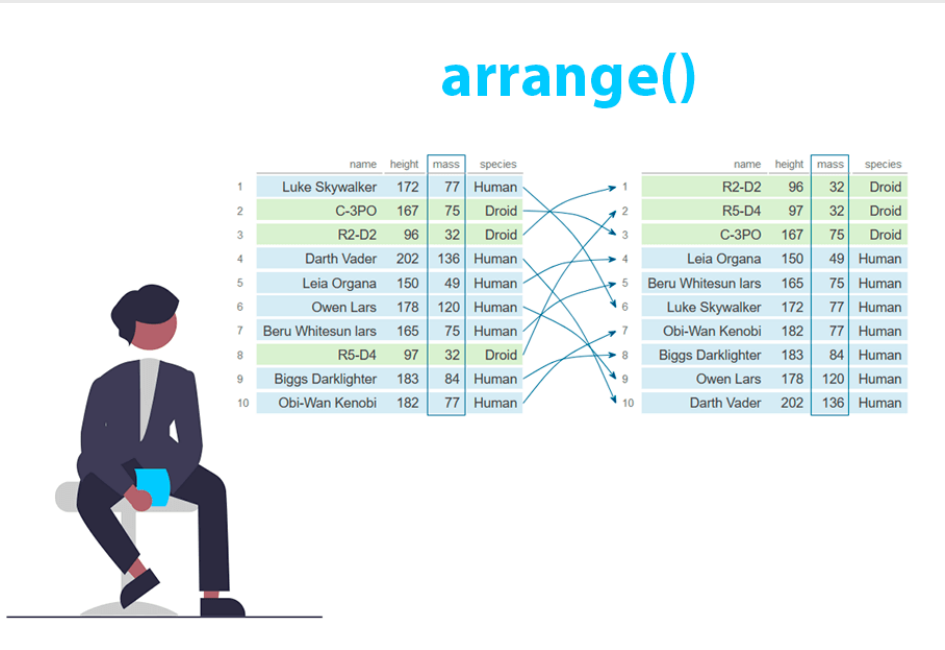
# E.g., sort by scheduled departure time (sched_dep_time), using departure delay (dep_delay)  # to break ties:

- arrange(flights, sched_dep_time, dep_delay)

# Use desc() on one of the reordering variables to sort by descending # order of that variable:

- arrange(flights, desc(sched_dep_time), dep_delay)

- Can reorder by as many variables as you like!



# select() and rename()

- select() allows us to subset the data by selecting the columns/variables that we're interested in. As before, the first argument is the tibble, and the subsequent arguments are the variables we wish to select.

# select only date-related variables (year, month, day):

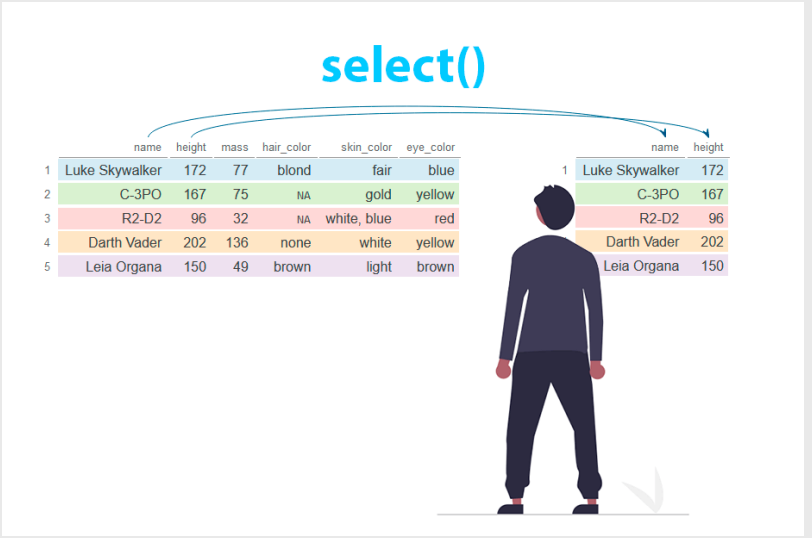
- select(flights, year, month, day)

#select all variables between year and day, inclusive

- select(flights, year:day)

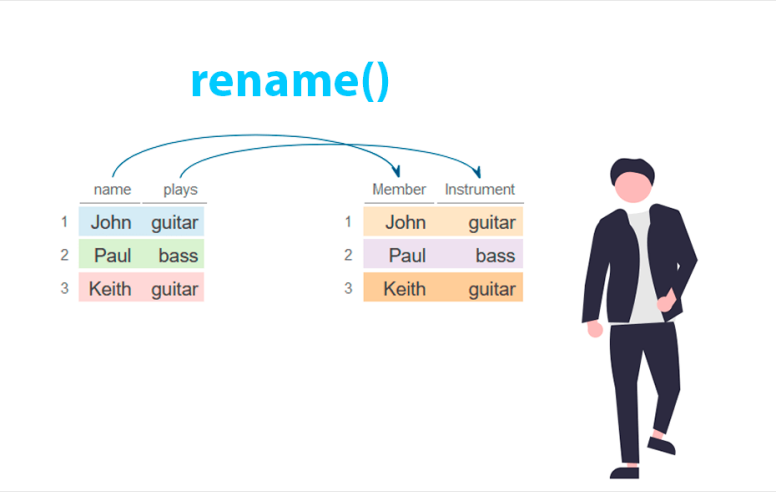
# select all variables except date-related variables:

- select(flights,-(year:day))



# select() and rename()

- Can also use select() to rename variables.
- Syntax is select(tibble, new.name = oldname).




# E.g., variable name tailnum doesn't match style of other
# variable names, so change it to match style:

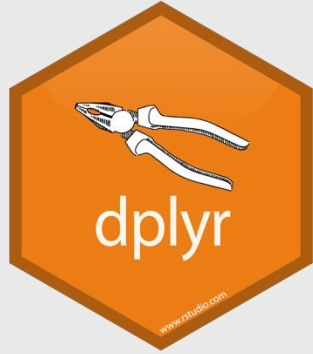
- select(flights, tail_num = tailnum)

- However, doing this produces a tibble with only the variable in question and drops the rest.

# Function rename() **renames variables without dropping**
**# the rest**. Syntax is the same:

- rename(flights, tail_num = tailnum)

# distinct()




- Helps extract unique values from a variable.
- Syntax: distinct(dataset,by=column1)

# E.g., variable name tailnum doesn't match style of other
# variable names, so change it to match style:

- select(flights, tail_num = tailnum)

- However, doing this produces a tibble with only the variable in question and drops the rest.

# Function rename() **renames variables without dropping the rest**.
# Syntax is the same:

- rename(flights, tail_num = tailnum)

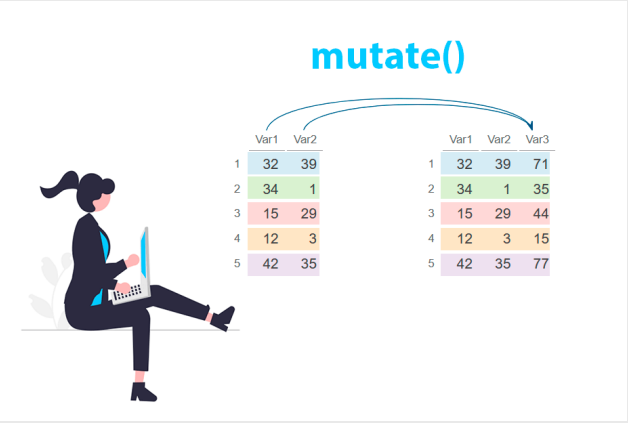
# mutate() and transmute()

- mutate() **adds new columns that are functions of existing columns**.
- Syntax: mutate(tibble, new_var = func_old_vars)

# E.g., add gain variable that gives amount of time exceeding scheduled time in the # air, and speed variable in mph:

- mutate(flights,
  gain = arr_delay - dep_delay,
  speed = distance/air_time * 60)

Hard to see right? What function could you add to see the data frame?




- Subsequent new variables in the same command can refer to previous new variables created.

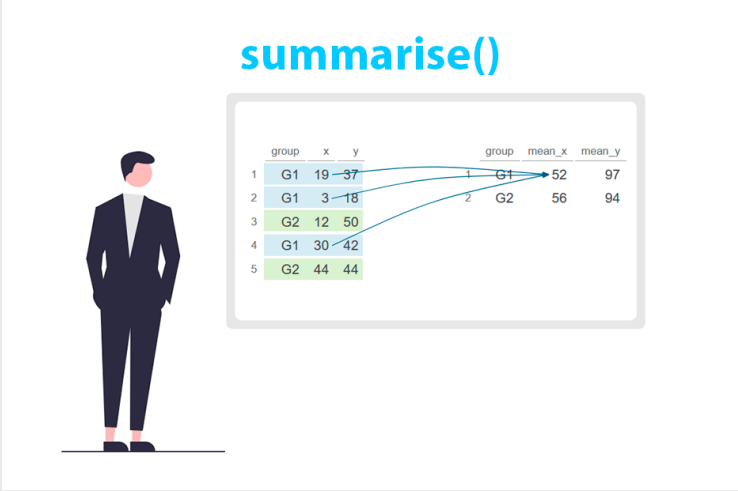
- If you only want to **keep the new variables** use transmute() with the same syntax.

# summarise()

- Collapses data frames to a single row.
- Syntax is summarise(tibble, entry_name=function)

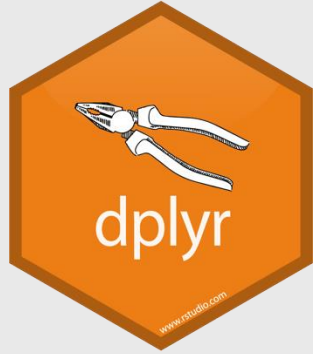
# E.g., determine mean and standard deviation of departure # delay (dep_delay):

```
summarise(flights,
          mean_dep_delay = mean(dep_delay, na.rm=T),
          sd_dep_delay = sd(dep_delay, na.rm=T))
```




- Alone this is useless until we touch upon the group_by() function.

# group_by()




- Groups data set into multiple subsets.
- Syntax is group_by(tibble, grouping_var).

- The resulting tibble includes the grouping variable in the tibble heading with the number of groups.

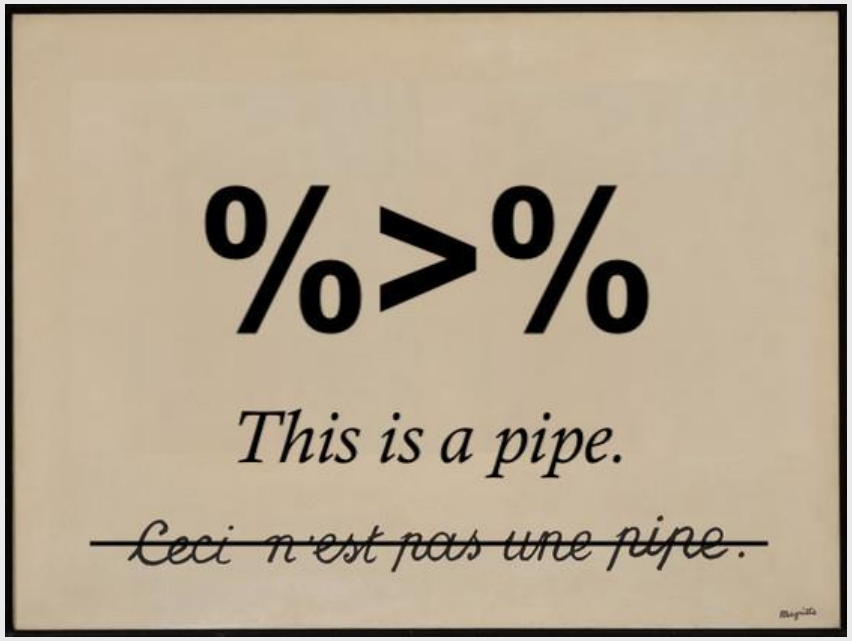
```
# E.g., group flights data by tail number (tailnum):
    by_tailnum = group_by(flights, tailnum)

    summarise(by_tailnum,
        mean_dep_delay = mean(dep_delay, na.rm=T),
        sd_dep_delay = sd(dep_delay, na.rm=T))
```

# Stringing together multiple dplyr commands…
# THE POWER OF PIPES


- All of these functions work fine on their own. But if we want to string together multiple dplyr commands and result in one tibble, we have to save our objects at each step of the way.

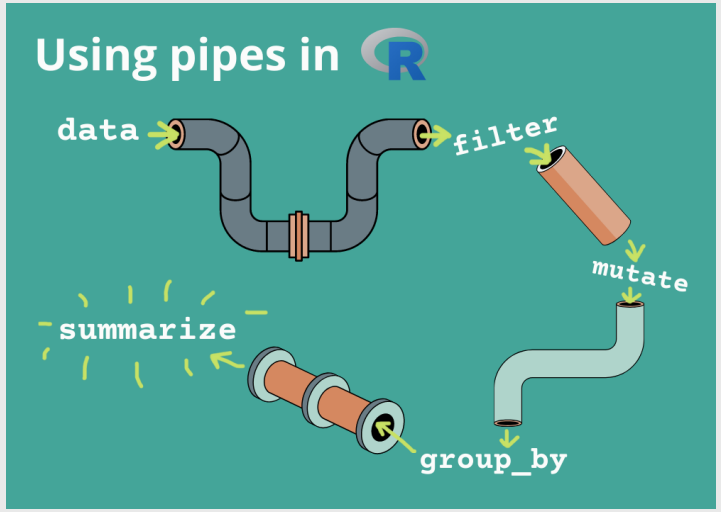
# create a flights tibble that displays only average departure and arrival delays for
# days on which either of those averages are greater than 30 minutes:

- A1 = group_by(flights, year, month, day)
- A2 = select(A1, arr_delay, dep_delay)
- A3 = summarise(A2, arr = mean(arr_delay, na.rm = T),
  dep = mean(dep_delay, na.rm =T))
- A4 = filter(A3, arr > 30 | dep > 30)



# Stringing together multiple dplyr commands…
# THE POWER OF PIPES


- **What's a pipe?**
  A forward-pipe operator that passes the output of one expression as the first argument to the next.
- **Why use pipes?**
  - Makes code linear and readable
  - Eliminates nested function calls
  - Encourages stepwise data transformation
- **Key package:**
  %>% comes from **magrittr** (and is re-exported by **dplyr**, **tidyverse**, etc.)



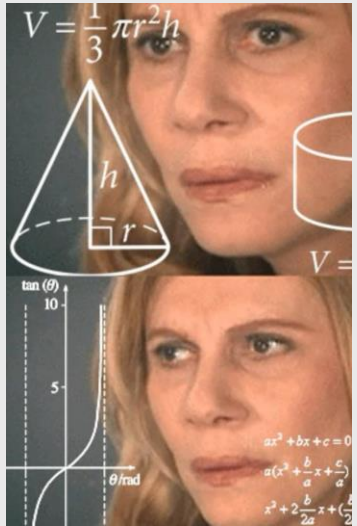
# Stringing together multiple dplyr commands… THE POWER OF PIPES

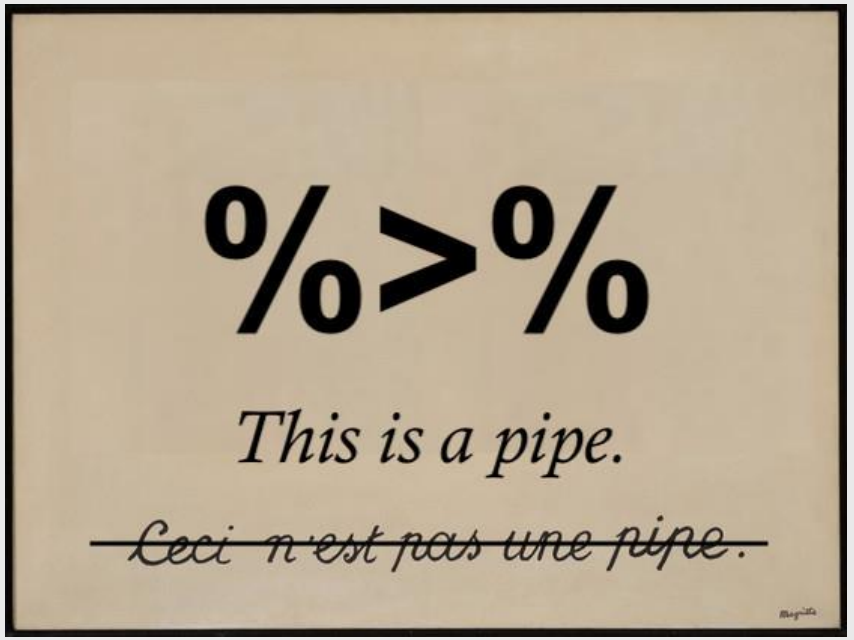
- All of these functions work fine on their own. But if we want to string together multiple dplyr commands and result in one tibble, we have to save our objects at each step of the way.

# create a flights tibble that displays only average departure and arrival delays for
# days on which either of those averages are greater than 30 minutes:

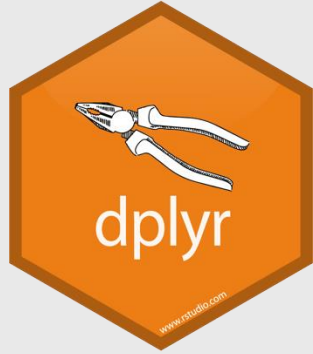
- A1 = group_by(flights, year, month, day)
- A2 = select(A1, arr_delay, dep_delay)
- A3 = summarise(A2, arr = mean(arr_delay, na.rm = T),
  dep = mean(dep_delay, na.rm =T))
- A4 = filter(A3, arr > 30 | dep > 30)

**This will clutter up your environment**



# Stringing together multiple dplyr commands




- Alternatively, we can write each command within the subsequent command:

```
# E.g., from the previous slide:
  flights %>%
 group_by(year, month, day) %>%
 select(arr_delay, dep_delay) %>%
summarise(
      arr = mean(arr_delay, na.rm = TRUE),
      dep = mean(dep_delay, na.rm = TRUE),
      .groups = "drop") %>%
 filter(arr > 30 | dep > 30)
```

**flights data frame**
**↓ group_by()**
**↓ select()**
**↓ summarize()**
**↓ filter()**

# Using AI

## Think of AI as…

- A “**calculator for words**” – it predicts the next token based on context, so more detail = better result.
- A paired programmer – don’t expect perfection up front; break problems into steps, discuss your approach, then iterate.

## Iterate & Refine

- Share your errors or unexpected outputs
- Explain “this is what I got vs. what I expected” to help the AI self-correct

At the end of the day if you don’t know the ‘R language’… Don’t expect AI to solve your problems