Intro to R for Biologists

IBiS Special Topics, Fall 2021
Class 1: Sept 23, 2021

Erik Andersen and Shelby Blythe

Who is this class for?

- Students with little or no experience with R, or those who have gotten rusty and would like a refresher.

- Students with a *need* for R in their research: ~1/3 of the class will be devoted to ‘personal project time’, this course will be most rewarding for students looking to build an analysis approach for their projects.

- Part of this course material used to be offered as a three-day intensive ‘boot camp’ to incoming IBiS students. By all accounts, retention was often low because “use it or lose it” very much applies to this material.

Who are your professors?

- We are both largely self-taught users of R and other programming languages.
  - (No long history of training in computational biology or computer science)

Who are you?

- What is your prior experience with R or any other programming language?

- What are your immediate goals for learning R?

The biggest challenge to starting to ‘code’

… is learning to think like a computer

- Example: we all know how to make a peanut butter and jelly sandwich, and think we could easily teach someone else how to do it.
  - What are the instructions?

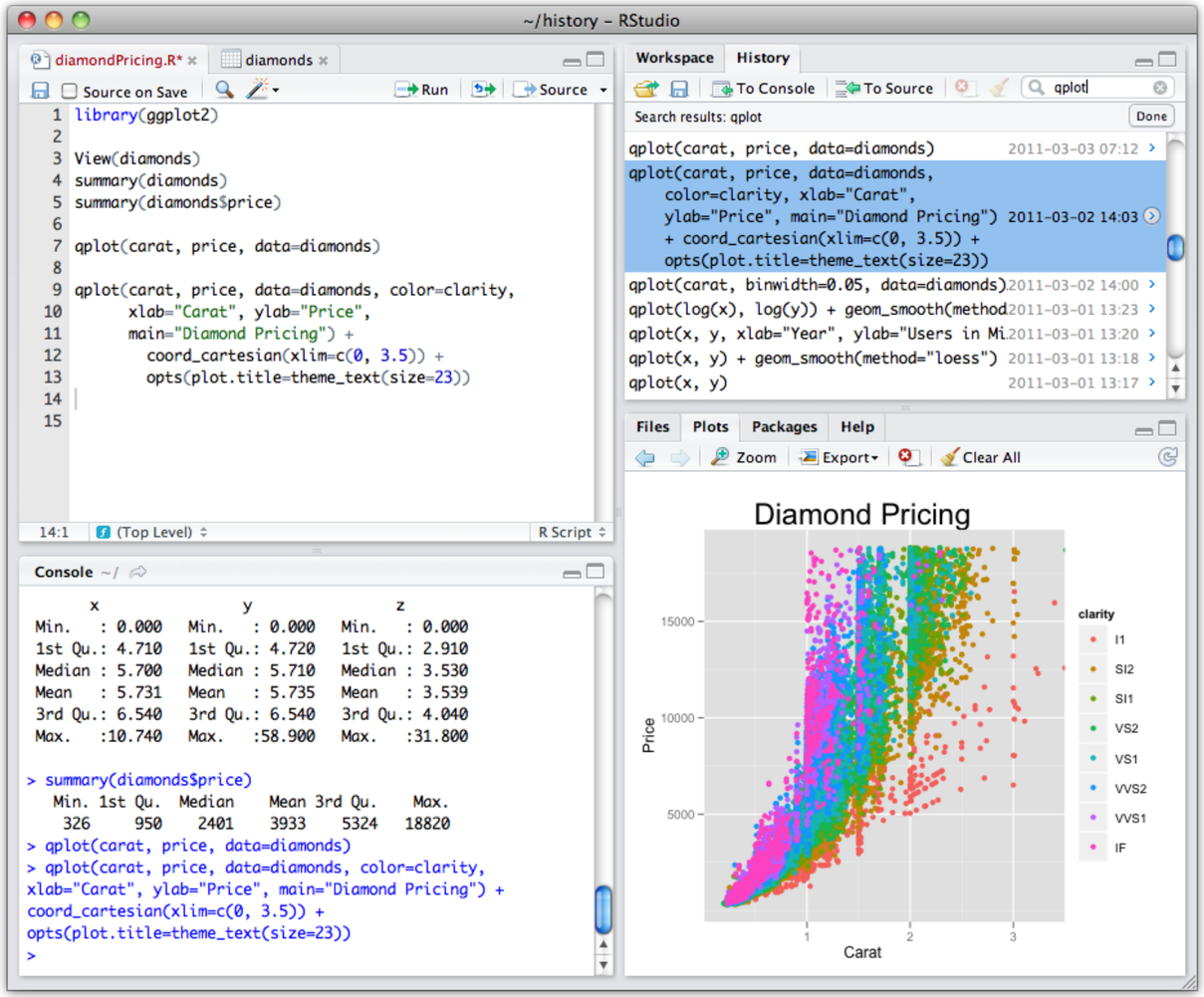
- “Learning to code” involves learning (or re-learning) how to write recipes under the functional and syntactical limitations imposed by the programming language. Computers only do what they are told and do not ‘read between the lines.”
- Inspired by this YouTube video: https://youtu.be/Ct-lOOUqmyY

First things first:

- Please ensure that you have set up your computer as described in the document “Computer_Setup.pdf”.
  - (Install: Command Line Tools, Homebrew, R, and RStudio).

RStudio makes  even easier!

Editor



Workspace

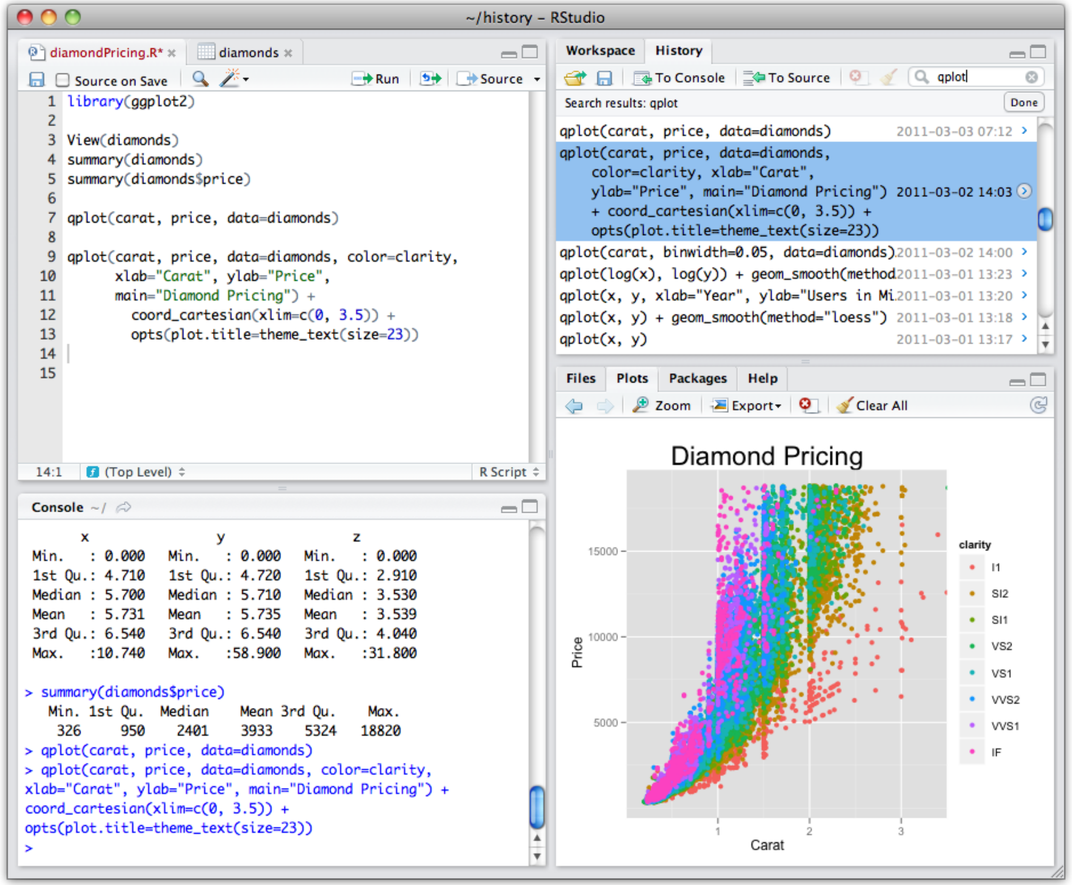
Console

Graphics

Two main ways to interact with R

Editor

Console

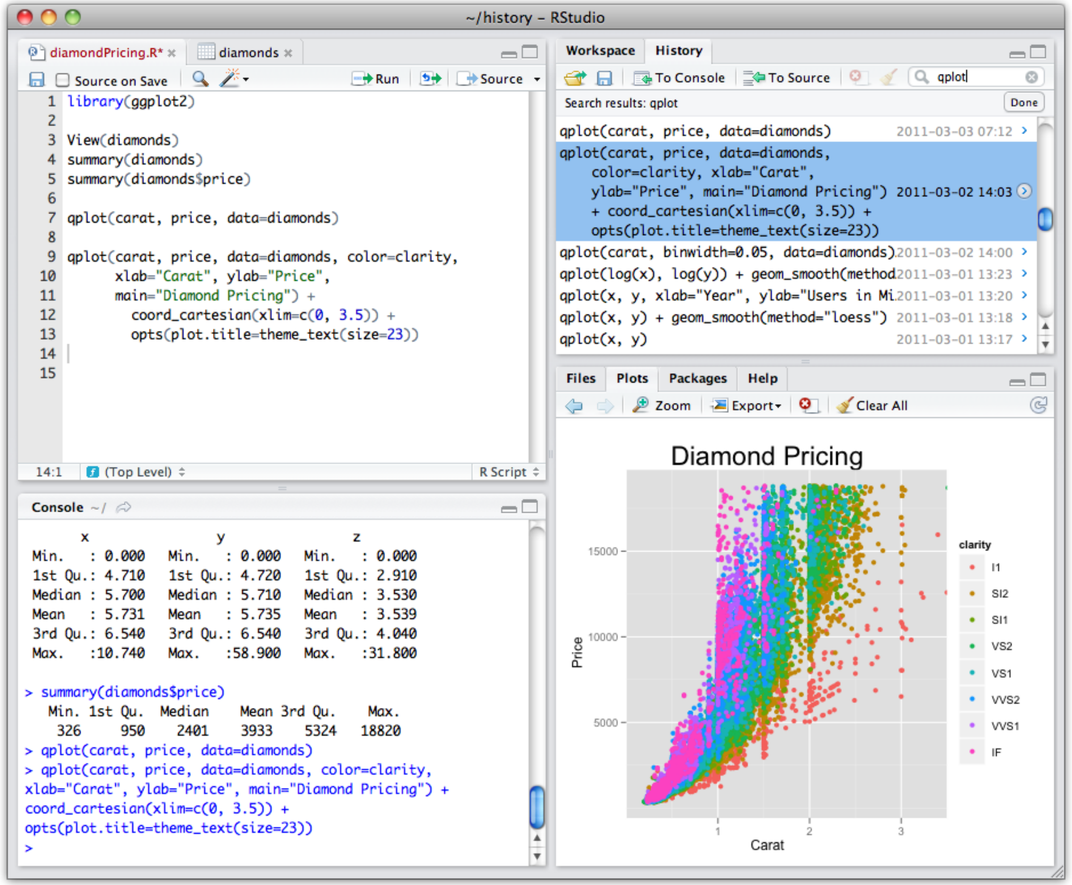


Think of the console as asking R questions about data.

Two main ways to interact with R

Editor

Console

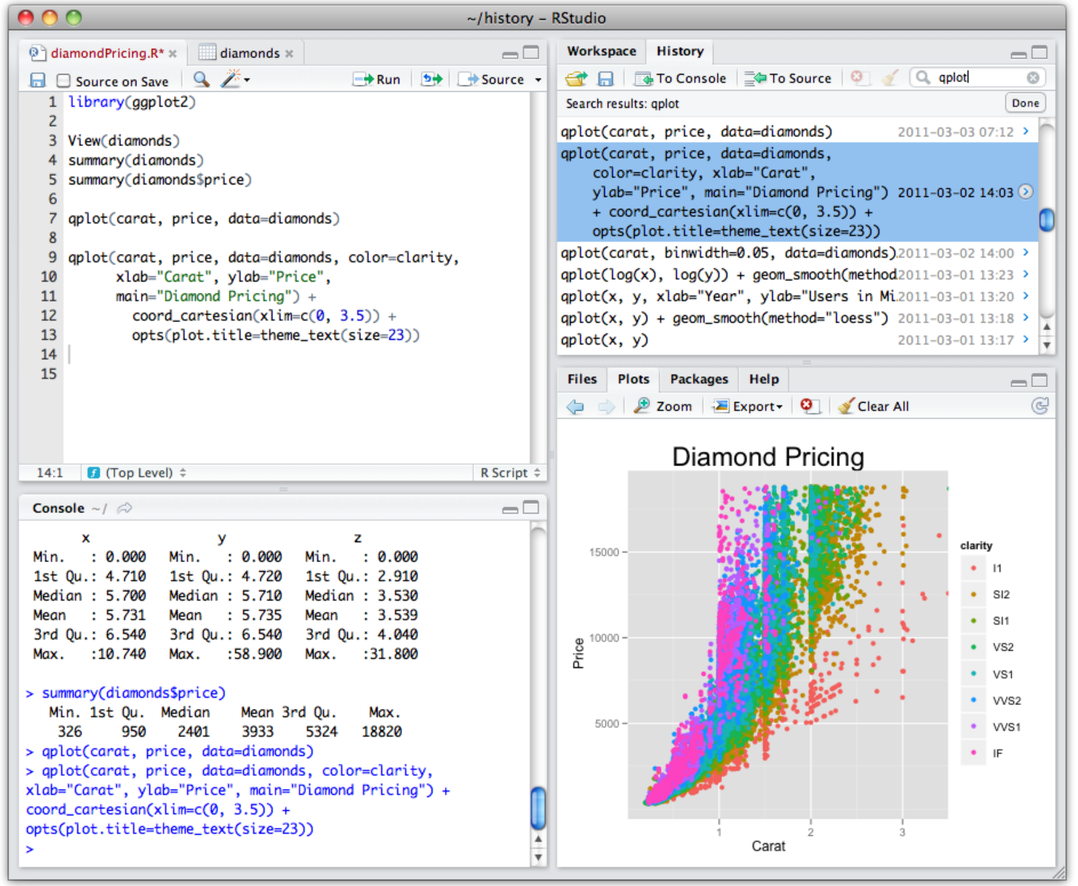


The editor is a plain text file editor for scripting and saving work (markdown too)

R helps you set up reproducible research practices

… but keeping track of all your work can be a challenge

- Part of the goal of this class is to teach you strategies for developing an analysis, and keeping track of a project as you would with a lab notebook.
- This starts with how you organize your file system, and how you name files.
- It extends to how you input commands into the editor, and learning to code in a predictable and reproducible way.



The standard should ultimately be: whatever code you write, that you could hand it to a stranger and it would run (and give the same results) on a completely different computer.

Very few people perform to this standard…

Basic R Syntax

- **>** The prompt.

  + at the prompt means that something isn't complete…

- **Functions** are words followed by (): i.e., `mean(x)` will calculate the mean of x.

- Equations are entered as **assignments**. There are two valid ways to assign the output of a command to a new variable.

  `y <- mean(x)` … method 1. `y = mean(x)` … method 2.

  (Both methods will assign the output of the function to a new object, y).

- Need **help**? Want to know the ins and outs of a function? Use the `?` function.

  `?mean` will take you to the help page for the `mean` function.

- Can't remember the exact name of the function? Try the `??` function.

- **#** is the **comment** character. Text following # is not evaluated by R, but can be used for notes.

Basic R Syntax: Explore

- In the RStudio console, assign 1 into y. Try both methods:

```r
# assign 1 into y (method 1)
y <- 1
y

# assign 1 into yy (method 2)
yy = 1
yy
```

- Assign 1 to 100 into y:

```r
# assign 1 to 100 into y
y = 1:100
y
```

- Add 1 and 6:

```r
# add 1 and 6
1 + 6
```

- Add 1 to y, assign to x:

```r
# add 1 to y, and assign it to x
x = 1 + y
```

- Plot the cube of y as a function of x:

```r
# plot x and the cube of y
plot(x, y^3)
```

Basic Data Types in R

Atomic vector types

character
numeric
integer
logical
complex
raw

atomic = only holds data of one type

- `y = 1`
  - (y is ‘numeric’) … try `class(y)`
- `y = “Erik”`
  - (y is character) … try `class(y)`
- `y = TRUE`
  - (y is logical) … try `class(y)`
    - (What is `y = “TRUE”`?)
    - (What about `y = True`?)

There are functions to *convert* between atomic data types:

as.logical(), as.character(), as.numeric()

Try:

```
y = TRUE
as.numeric(y)

y = 1
as.character(y)

y = “Erik”
as.logical(y)
```

… sometimes conversion yields nonsense.

Basic Data Types in R

- It is common to need to work with long *vectors* of values. We already saw this with the command:

```
y = 1:10
```

  - This assigns to y the whole numbers from 1:10.

The function you will use most often is **c()**. **c** stands for *combine.*

Try:

```
y = c(1:10)
y = c(TRUE, TRUE, FALSE)
y = c("Erik", "Shelby", "Rich")
```

c can be used also to append values to one another.

Try:

```
y = c("Erik", "Shelby", "Rich")
x = c("Sadie", "Keara", "Carole")
z = c(y, x, "Curt")
```

Sometimes, you want a regular sequence of numbers. **seq()** is more versatile than the prior example.

Try:

```
y = seq(from = 1, to = 10, by = 0.5)
```

And sometimes you want a set of random numbers. (There are lots of ways to do this, but **runif()** and **rnorm()** are good to memorize).

Try:

```
y = runif(10, min = 0, max = 10)
```

(Try it a few times… are the results the same?)

Now, try:

```
set.seed(123)
y = runif(10, min = 0, max = 10)
```

(Repeat these two lines a few times. What do you see?)

Basic Data Types in R

- But once you have long vectors of numbers, you need to be able to access only a subset of them.
- This is achieved by using *bracket notation.*

R has a built-in function **LETTERS** for giving you “all the capital letters” that we can assign to a variable, L.

Try:

```
L = LETTERS
```

Using **brackets [ ]**, we can pass the *index* of an element of L that we are interested in getting.

Try:

```
L[1]
L[17]
L[1:5]
L[27]
```

One of the most powerful things about using a program like R is that you can get values from a vector that fulfill certain criteria. This is done by combining **brackets** with **logical indexing.**

First, create a vector of the numbers from 1:26

```
x = c(1:26)
```

*We can perform **logical operations** on such a vector. Like, which elements are greater than 12?*

Try:

```
x > 12
```

*The output of that operation is a **logical vector**. It can be used to subset a completely different vector if it is placed within **brackets.***

Try:

```
L[x > 12]
```

Basic Data Types in R

- There is a defined syntax for performing logical operations. These are worth committing to memory.

> greater than (>= greater than or equal to)

< less than (<= less than or equal to)

== equal to

!= not equal to

& and

| or

! (Inverts the logical vector)

%in% is a member of

Using our **LETTERS** assigned to L and the index vector x:

Try:

```
L[x < 5]
L[x <= 5]
```

Now try:

```
L[x < 5 | x > 24]
L[x < 5 & x > 24]
```

(You will spend some time thinking deeply about the difference between **and** and **or**).

It helps to think of the **logical operation** in the **bracket** as a completely independent function, with its own output. The **bracket** only returns indices of values of that interior function that evaluate to **TRUE.**

Try:

```
x > 3
!x > 3
```

Now try:

```
L[x < 3]
L[!x < 3]
```

When combining logical operations with **&** and **|**, it can help to use parenthesis.

Try:

```
L[x < 3 | x > 23]
L[!x < 3 | x > 23]
L[!(x < 3 | x > 23)]
```

Basic Data Types in R

- There is a defined syntax for performing logical operations. These are worth committing to memory.

> greater than (>= greater than or equal to)

< less than (<= less than or equal to)

== equal to

!= not equal to

& and

| or

! (Inverts the logical vector)

%in% is a member of

- When performing logical operations on **character** vectors it is helpful to use the operator **%in%.** Say you have a list of genes:

```
genes = c('Nanog', 'Klf4', 'Sox2', 'Oct4', 'Myc')
```

- And you want to know whether "Oct4" is a member.

Try:

```
genes == "Oct4"
```

Now try:

```
genes %in% "Oct4"
```

However:

```
genes == c("Oct4", "Klf4")
genes %in% c("Oct4", "Klf4")
```

But. (This is important)

Try:

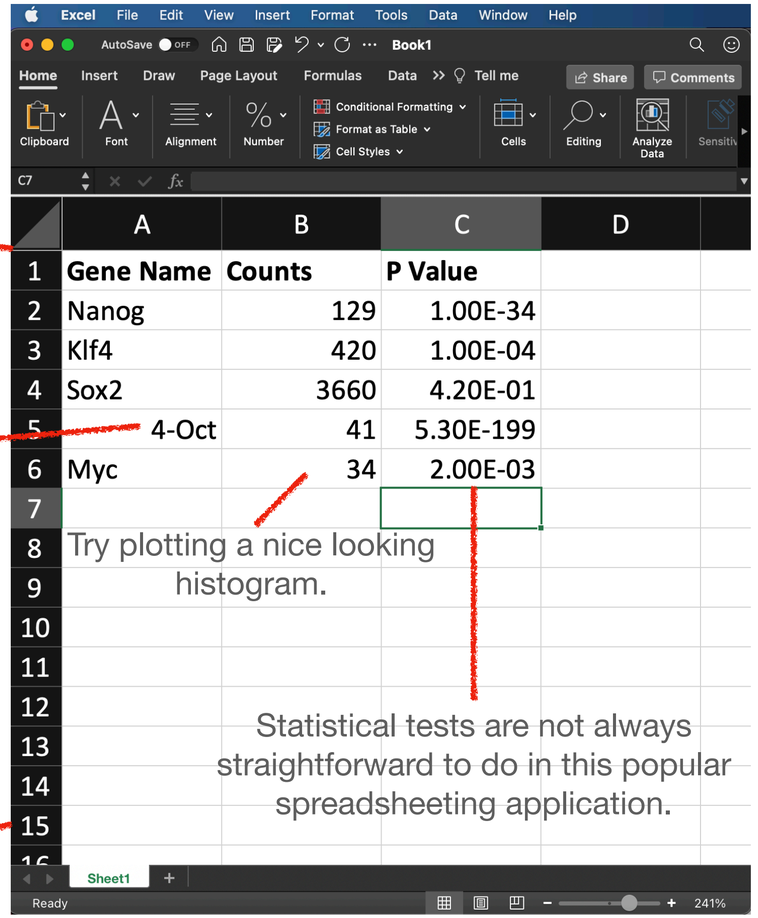
```
"Oct4" %in% genes
```

Bottom line, use **%in%** to ask whether certain character strings are members of a larger set of character strings.

But most data are not single vectors.

We deal with tables. But for most folks, tables = spreadsheets

- Spreadsheets are mostly fine for small data tables.
  - What happens when they get big?
  - What happens when you need to extract subsets of data?
  - What happens when you have to do a repetitive task?
- **This is where R can help.**



You are restricted to this horrible rectangular space

This is a date now for some reason.

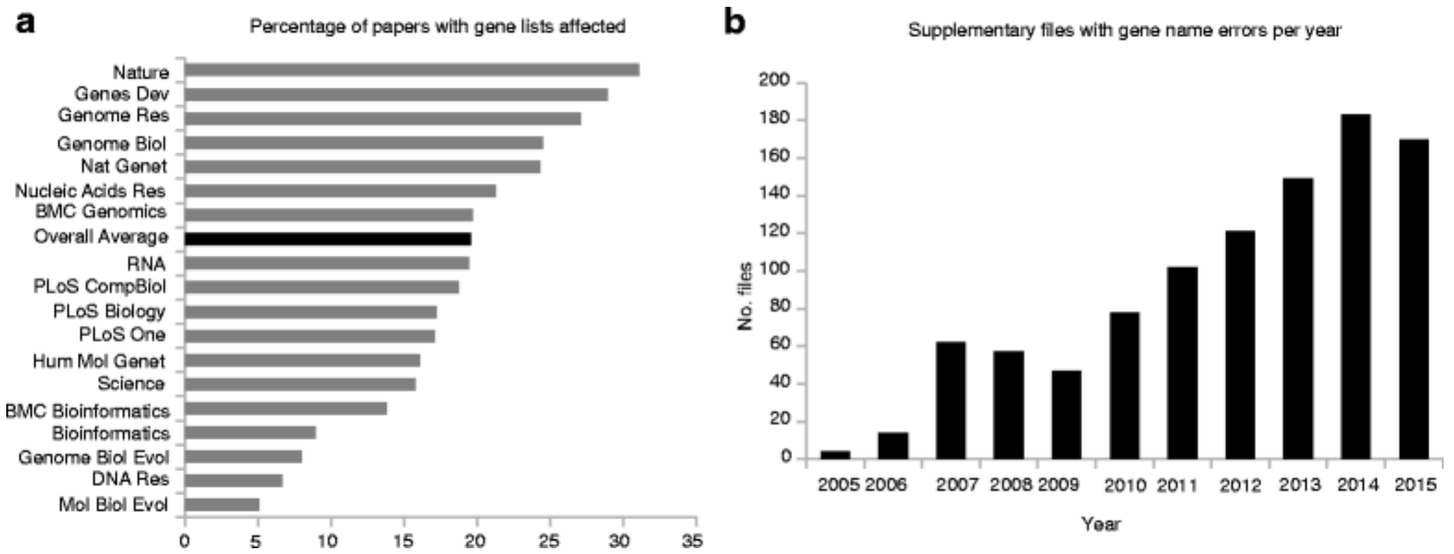
Try plotting a nice looking histogram.

Statistical tests are not always straightforward to do in this popular spreadsheeting application.

Big data sets can have millions of rows (limited here to ~$1x10^6$)

Microsoft Excel is a menace

The *Oct4* to “October 4th” transformation is a frequent error in the scientific literature



Gene name errors are widespread in the scientific literature
Ziemann, Eren, and El-Osta. Genome Biology August 2016

Microsoft Excel is a menace

Incredibly, this has led scientists to change the names of genes just to accommodate this software!

MICROSOFT \ REPORT \ SCIENCE \

Scientists rename human genes to stop Microsoft Excel from misreading them as dates

Sometimes it's easier to rewrite genetics than update Excel

By James Vincent | Aug 6, 2020, 8:44am EDT

SHARE

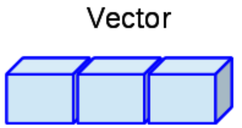
The Verge: Aug 6, 2020.
https://www.theverge.com

Oct4: is now Pou5f1
Sept7: is now Septin7
Mar1: is now RTL1
March1: is now MarchF1

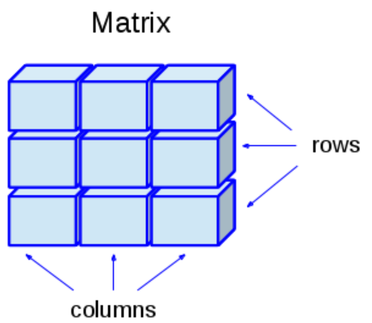
They also 'fixed' gene names that were potentially offensive (but Shh is still in there).

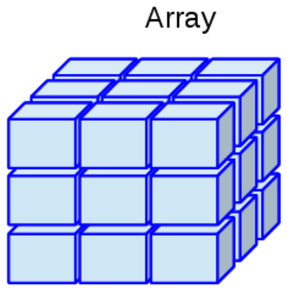
Data Structures in R

- **Vectors** are one-dimensional datasets of *any one data type.*

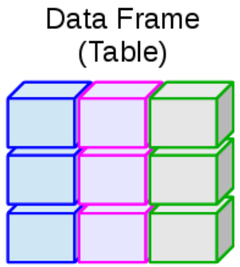


- **Matrices** and **Arrays** are two- and three-dimensional datasets of *any one data type.*

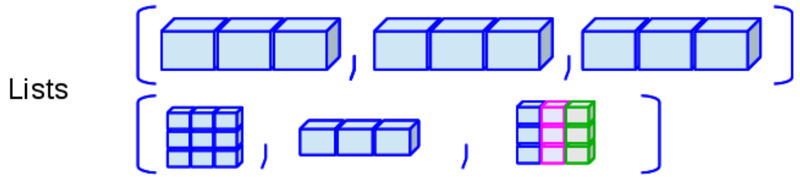




- **Data Frames** are two-dimensional data sets of *any data type*.



- **Lists** groups of various data types including other lists.

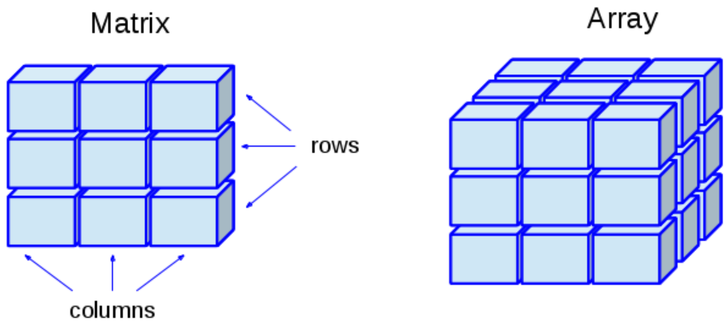


Data Structures in R

Matrices (and Arrays)

- You can think of matrices and arrays as 2D and 3D collections of vectors, arranged in rows + columns.

- These structures hold only one type of data.



To explore matrices, let's make a toy example with random data. For example, 25 random numbers (rounded):

```
set.seed(2021)
filler = round( runif(25, min = 1, max = 100))
my.mat = matrix(filler, nrow = 5, ncol = 5)
```

```
> set.seed(2021)
> filler = round( runif(25, min = 1, max = 100))
> my.mat = matrix(filler, nrow = 5, ncol = 5)
> my.mat
     [,1] [,2] [,3] [,4] [,5]
[1,]   46   70    4   26   15
[2,]   79   64   84   51   96
[3,]   71   27   61   87   40
[4,]   39   82   57   96   28
[5,]   64   98   82   55   58
```

Individual elements of a matrix are also accessed through **brackets**, but in two different ways.

Each element has an *index,* (i.e., the 1st, 2nd, … *nth* element), designated column-wise. Sometimes this is useful.

```
> my.mat[2]
[1] 79
> my.mat[3]
[1] 71
```

More often, however, you will want to specify the element you want more precisely. This can be done in **brackets** by specifying the [ROW, COLUMN] coordinates.

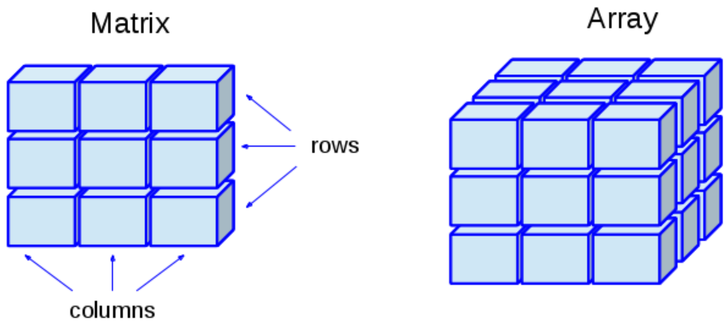
```
> my.mat[2,1]
[1] 79
```

Data Structures in R

Matrices (and Arrays)

- You can think of matrices and arrays as 2D and 3D collections of vectors, arranged in rows + columns.

- These structures hold only one type of data.



continued…
Using **brackets** and [ROW, COLUMN] positions, you can easily specify either a single value, or get an entire row (or column) of values.

```
> my.mat[3,3] # gets the value in the 3rd row, 3rd column
[1] 61
> my.mat[3,] # gets all values from the 3rd row
[1] 71 27 61 87 40
> my.mat[,3] # gets all values from the 3rd column
[1]  4 84 61 57 82
```

The rows and columns (or the whole matrix) can also be queried by **logical operations.**

You have to use [row, column] notation appropriately to do this, however.

Try:

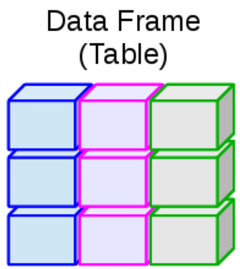
```
my.mat > 50
```

```
my.mat[,1] > 50
```

Data Structures in R

Data Frames

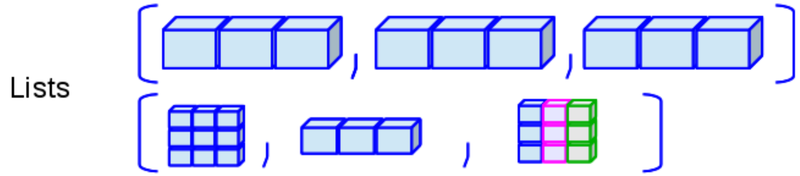
- By far the most useful and common data structure in R is the Data Frame. It is like a matrix, but it holds different classes of data (column-wise).



- We will explore data frames as part of the post-lecture exercises.

Lists

- Lists are collections of different data structures. We will introduce lists later in the course (but it is good to know now that they exist).



In general, lists are good for collecting sets of related data, and R includes specialized functions that operate over lists.